University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

# DIPLOMA THESIS

Pilsen, 2007

Ivan Habernal

University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

# Diploma Thesis

# Lexical Class Analysis

Pilsen, 2007 Ivan Habernal

## Acknowledgments

I would like to thank Ing. Miloslav Konopík for his supervising and Mgr. Alice Tihelková, Ph.D. for her help.

I hereby declare that this diploma thesis is completely my own work and that I used only the sources cited.

Pilsen, May 21, 2007 Ivan Habernal, .....

# Abstract

**Lexical Class Analysis**

The goal of the thesis is to explore and develop a set of methods intended for lexical class identification. The identification of lexical classes is the first step of stochastic semantic parsing. The methods are based on context-free grammars and parsing techniques. The implementation uses the Java Speech API and the Java Speech Grammar Format for grammar processing.

**Keywords**: lexical classes, stochastic semantic parsing, natural language processing, grammars, JSAPI, JSGF.

# Contents

# Chapter 1

# Introduction

The human-computer communication is one of the most challenging topics of Artificial Intelligence (AI). Since speech is the most natural and the most common way of human communication, there is a need for using it as a communication medium between human and computer.

*Natural Language Processing* (NLP) is a branch of AI and linguistics which deals with the problems of automated generation and understanding of natural human language. Applications of this technology include translation, information retrieval, categorization, knowledge extraction and dialog systems. Recent trends in NLP are heading towards stochastic processing of natural language.

The process of extracting meaning from an utterance by computer and storing the meaning in the computer model is called *Natural Language Understanding* (NLU). *Semantic analysis* is the first step of the NLU process. The goal of semantic analysis is to represent what the subject intended to say. *The identification of lexical classes* is a fundamental step of semantic analysis.

In the thesis, we focus on the possibilities of using *context-free grammars* and *parsing techniques* to identify the *lexical classes* within an utterance.

## 1.1 Structure of the thesis

The thesis is divided into two main parts — the *theoretical basis* part and the *practical realization* part. The structure of thesis is as follows:

- In the second chapter, the *formal grammars* are introduced. The context-free grammars and their properties are emphasized.
- The third chapter deals with *parsing*. The main general parsing approaches are discussed, together with parsing techniques with limits to grammar type.

- The use of the *Java™ Programming Language* in the field of NLP is discussed in chapter 4. Two main technologies are presented — the Java Speech API and the Java Speech Grammar Format.
- Chapter 5 provides a short introduction to *Stochastic Semantic Parsing*.
- In Chapter 6, the implementation of an abstract annotation editor is introduced.
- Chapter 7 is concerned with the implementation of Java Speech API. Our own implementation will be introduced alongside the existing libraries.
- Chapter 8 describes the implementation of a bottom-up active chart parser.
- Chapter 9 describes the main task of this thesis — the application intended for *lexical class identification*. The chapter contains the implementation description and the discussion of experimental results.
- The conclusion is provided in Chapter 10. This chapter also discusses the future work.

# Part I

# Theoretical basis

# Chapter 2

# Grammars

There are a number of ways of representing a language. For example, methods based on *regular expressions* can be used [AF98]. In this chapter we focus on perhaps the most useful and general means of language description – *grammars*.

There are two categories of formal grammars – *generative grammars*, which are sets of rules for how strings in a language can be generated, and *analytic grammars*, which are sets of rules for how a string can be analyzed to determine whether it is a member of the language. In short, an analytic grammar describes how to recognize when strings are members in the set, whereas a generative grammar describes how to write only those strings in the set [AF98].

In linguistics, the use of generative grammars is preferred since language can be defined as a set of strings generated by the grammar. Therefore, only generative grammars will be discussed below. We will start with definitions of grammar and its relation to language. Later, the grammar hierarchy will be described.

## 2.1 Grammar definition and characteristics

**Definition 1** *A* grammar *is a quadruple* $(\Sigma, V, S, P)$*, where*

$\Sigma$ *is a finite non-empty set called* terminal alphabet*. The elements of this set are called* terminal symbols *or* terminals*.*

$V$ *is a finite non-empty set disjoint from* $\Sigma$ *containing* nonterminal symbols *(also called* nonterminals*).*

$S \in V$ *is a distinguished symbol called* starting symbol*.*

$P$ *is a finite set of* production rules*, each of the form*

$$\alpha \rightarrow \beta$$

*where* $\alpha \in (\Sigma \cup V)^\star V (\Sigma \cup V)^\star$ *and* $\beta \in (\Sigma \cup V)^\star$. *Each production rule maps from one string of symbols to another, where the first string contains at least one nonterminal symbol.*

**Definition 2** *Let* $G = (\Sigma, V, S, P)$ *be a grammar. A* sentential form *of* $G$ *is any string of terminals and nonterminals.*

**Definition 3** *Let* $G = (\Sigma, V, S, P)$ *be a grammar, and let* $\gamma_1$, $\gamma_2$ *be two sentential forms of* $G$. *Then* $\gamma_1$ directly derives $\gamma_2$ *(written* $\gamma_1 \Rightarrow \gamma_2$*), if* $\gamma_1 = \sigma\alpha\tau$, $\gamma_2 = \sigma\beta\tau$ *and* $\alpha \to \beta$ *is a production rule in* $P$.

**Definition 4** *Let* $G = (\Sigma, V, S, P)$ *be a grammar, and let* $\gamma_1$, $\gamma_2$ *be two sentential forms of* $G$. *Then* $\gamma_1$ derives $\gamma_2$ *(written* $\gamma_1 \Rightarrow^\star \gamma_2$*), if there exists a sequence of sentential forms* $\sigma_1$, $\sigma_2$, *such that*

$$\gamma_1 \Rightarrow \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_2 \Rightarrow \gamma_2$$

*and is called* derivation *of* $\gamma_2$ *from* $\gamma_1$.

**Definition 5** *Let* $G = (\Sigma, V, S, P)$ *be a grammar. The* language generated by $G$, *denoted by* $L(G)$, *is defined as*

$$L(G) = \{x | x \in \Sigma^\star, S \Rightarrow^\star x\}.$$

**Example** Having the grammar $G_1$ with $\Sigma = \{0, 1\}$, $V = \{S, T, O, I\}$, starting symbol $S$ and

$$P = \{S \to OT, \quad S \to OI, \quad T \to SI, \quad O \to 0, \quad I \to 1\},$$

we can see that the language generated by this grammar can contain only symbols 1 and 0 in form

$$L(G_1) = \{0^n 1^n | n \geq 1\}.$$

**Definition 6** *Grammar is* left-recursive *if it contains at least one nonterminal* $A$, *such that*

$$A \Rightarrow^\star \alpha A \beta \qquad \textit{and} \qquad \alpha \Rightarrow^\star \epsilon,$$

*where* $\alpha$ *is nonterminal and* $\beta$ *is nonterminal or terminal.*

## 2.2 Chomsky Hierarchy

Grammars can be divided into four classes by increasing the restrictions on the form of the production rules. Such a classification is called the *Chomsky hierarchy* [Cho56].ž

**Type-0 grammars** – also called *unrestricted grammars*. This level includes all formal grammars. These grammars generate all languages that can be recognized by *Turing machine*.

**Type-1 grammars** – also called *context-sensitive grammars*. These grammars have each production rule in form

$$\alpha A \beta \rightarrow \alpha \gamma \beta,$$

where $A$ is nonterminal and $\alpha$, $\beta$ and $\gamma$ are strings of terminals or nonterminals. Strings $\alpha$ and $\beta$ can be empty, but $\gamma$ cannot. Rule

$$S \rightarrow \epsilon,$$

where $\epsilon$ represents an empty string, is allowed only if $S$ does not appear on the right side of any production rule.

**Type-2 grammars** – also called *context-free grammars*. Production rules of these grammars are in form

$$A \rightarrow \gamma,$$

where $A$ is a single nonterminal and $\gamma$ is a string of terminals or nonterminals. These grammars generate *context-free languages* that can be recognized by a non-deterministic pushdown automaton.

**Type-3 grammars** – also called *regular grammars*. Each production rule has one of the following three forms:

$$A \rightarrow \gamma B, \quad A \rightarrow \gamma, \quad A \rightarrow \epsilon,$$

where $A$ and $B$ are nonterminals and $\gamma$ is terminal; $A$ and $B$ can be equal. The third rule is allowed only if $A$ does not appear on the right side of any rule. The first production rule represents the *right-linear grammar*. Production rules of *left-linear grammars* are in form

$$A \rightarrow B\gamma$$

For any left-linear grammar there exists a right-linear grammar which generates the same language. Languages generated by these grammars can be recognized by a finite state automaton.

Every regular grammar is a context-free grammar, but not every context-free grammar is context-sensitive.

### 2.2.1 Grammar equivalence

Since formal language is defined as a set of strings, we could expect two grammars to be equivalent if they generate the same language. Actually, we distinguish two kinds grammar equivalence.

**Strong equivalence** We say that two grammars are strongly equivalent if they generate the same set of strings and if they assign the same phrase structure (syntactic tree) to each generated set of strings (sentence).

**Weak equivalence** We say that two grammars are weakly equivalent if they generate the same set of strings but assign different phrase structure to each sentence.

### 2.2.2 Normal forms

Every context-free grammar can be transformed into an equivalent grammar with all production rules in form

$$A \to BC \quad \text{or} \quad A \to c,$$

where $A$, $B$ and $C$ are nonterminals (possibly equal) and $c$ is terminal. We can also include an empty string if it is present in $L(G)$ by including the $S \to \epsilon$ rule. This form is called *Chomsky normal form*.

Another form is called *Greibach normal form*. A grammar in that form must have all production rules in form

$$A \to \gamma B \quad \text{or} \quad S \to \epsilon,$$

where $A$ is nonterminal, $\gamma$ is terminal and $B$ is a (possibly empty) sequence of nonterminals excluding the starting symbol $S$. Every context-free grammar can be transformed into an equivalent grammar in *Greibach normal form* as well [BK99].

## 2.3 Context Free Grammars

Context-free grammars (CFG) are very important for NLP. First, the formalism is powerful enough to describe most of the structures in natural language. Second, it is still restricted enough so that efficient parsers for analyzing sentences can be built [All95]. The parsers will be discussed in chapter 3.

An alternative and equivalent definition of context-free languages employs non-deterministic push-down automata: a language is context-free if and only if it can be accepted by such an automaton. The proof of this theorem can be found in [AF98].

### 2.3.1 String derivation

A string can be derived from the start symbol of context-free grammar in two common ways. The first strategy is replacing the leftmost nonterminal first. This approach is called the *leftmost derivation*. Let us have grammar $G_1$ defined as

$$\begin{aligned} (1) \quad & S \to S + S \\ (2) \quad & S \to 1 \\ (3) \quad & S \to a \end{aligned}$$

and the string `1 + 1 + a`. Then the left derivation of this string, as a sequence of applied rules, is $\{1, 1, 2, 2, 3\}$.

Another strategy, called *rightmost derivation*, would produce the list of rules – $\{1, 3, 1, 2, 2\}$. The approach is to replace the rightmost nonterminal first.

Sentences (strings) that can be derived by a grammar are in the formal language defined by that grammar, and are called *grammatical* sentences. Otherwise, sentences that cannot be derived by a grammar are called *ungrammatical*.

### 2.3.2 Ambiguity

*Syntactic tree* is a tree structure that describes how the production rules have been applied to generate a given string $w$ of terminals. The root of the tree is the starting symbol $S$ of the grammar, the leaves are terminal symbols that make up the string $w$ and the branches of the tree describe how the production rules are applied.

**Definition 7** *Let $G$ be a grammar. We say that the sentence $w$ generated by the grammar $G$ is* ambiguous *if there are two different syntactic trees for the sentence $w$. A grammar is* ambiguous *if it generates at least one ambiguous sentence. The grammar is* non-ambiguous *otherwise.*

Consider the following context-free grammar $G$ with production rules[1]

$$S \to S + S, \qquad S \to S - S, \qquad S \to a$$

For the input string `a + a - a` there are two leftmost derivations (and corresponding syntactic trees) as shown in figure 2.1. The language that it generates, however, is not necessarily ambiguous. The following is a non-ambiguous grammar generating the same language:

$$S \to S + a \mid S - a \mid a.$$

[1] Mostly, the shorter form of writing rules is used, which would be for our example $S \to S + S \mid S - S \mid a$

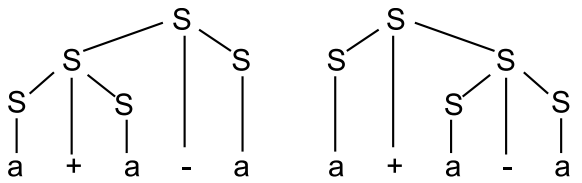


Figure 2.1: Two syntactic trees for ambiguous grammar

The general question of whether a grammar is ambiguous is undecidable. Since the *halting problem*[2] can be encoded as an ambiguity problem, and the halting problem cannot be decided [MSL03], then there exists no algorithm to determine the ambiguity of a grammar.

### 2.3.3 Properties of CFG

**Definition 8** *A class of languages is said to be* closed under *a particular operation (such as union, intersection, complementation, concatenation or Kleene closure) if every application of the operation on language of the class produces a language of that class.*

Having this definition, we can describe the following properties of context-free grammars.

- The *union* and *concatenation* of two context-free languages is context-free, but the *intersection* need not be.
- The *reverse* of a context-free language is context-free, but the complement need not be.
- The *intersection* of a context-free language and a regular language is always context-free.

We should also mention the fact that determining whether a context-sensitive grammar describes a context-free language is undecidable [AF98].

### 2.3.4 Probabilistic context-free grammars

Probabilistic context-free grammar (PCFG), also known as the Stochastic context-free grammar (SCFG), is a simple augmentation of the CFG.

Let $G = (\Sigma, V, S, P)$ be a grammar (see definition 1). Each production rule from $P$ is augmented with such a conditional probability $p$ that

$$A \rightarrow \beta \ [p],$$

[2]The halting problem is a decision problem which can be informally stated as follows: Given a description of a program and a finite input, decide whether the program finishes running or will run forever, given that input. [MSL03]

where $0 \leq p \leq 1$ and

$$\sum_{i=1}^{j} p_i = 1$$

for all production rules $A_i \rightarrow \beta_j$.

Thus, a PCFG is a 5-tuple $G = (N, \Sigma, P, S, D)$ where $D$ is a function assigning probabilities to each rule in $P$. This function expresses the probability $p$ that the given nonterminal $A$ will be expanded to the sequence $\beta$.

# Chapter 3

# Parsing

We can define parsing as a task of *recognizing a sentence and assigning a syntactic structure to it*. This section focuses on parsing context-free grammars. Since CGFs are a declarative formalism, they do not specify the process of computing and creating a parse tree for given sentences. Therefore, there exist many possible algorithms for automatically assigning a context-free tree to an input sentence.

As mentioned in section 2.3.1, parse trees are directly useful in grammar checking – for *grammatical sentences* there exists a parse tree of the given grammar. In addition, parsing is an important intermediate stage of representation for *semantic analysis*. Finally, the stochastic versions of parsing algorithms are parts of *speech recognizers* and both stochastic and non-stochastic versions are incorporated into *language models* [JM00].

In this chapter two basic approaches to parsing will be described – the *top-down* and the *bottom-up* parsing strategy. Simple algorithms will be introduced as well as the more efficient parsers using a *chart* structure.

## 3.1 Parsing as a search problem

We can consider parsing as a search problem. Then, the goal of that search is to find all trees whose roof is the start symbol S of a grammar. The root covers the words in the input sentence.

Thus, there are two constraints on such search. First, the parse tree must have the same count of leaves as the count of input words and, second, the root must always be the starting symbol $S$. These two constraints bring two different parsing strategies – *top-down parsing* (also called *goal-directed parsing, search*) and *bottom-up parsing* (also called *data-directed parsing, search*).

## 3.2 Basic Top-down Parser

A *top-down parser* searches for the parse tree by trying to build the tree from the root node containing starting symbol $S$ and derivate it down to the terminals.

### 3.2.1 Algorithm description

The basic algorithm is based on search strategy called *backtracking*. It uses a stack (or *agenda*) of states, where each state is represents the so-far parsed tree. It can be formalized as follows:

- Add initial $S$ tree to the agenda.
- In the main loop:
  - Try to expand the current tree with leftmost (or rightmost) rule and add it into agenda.
  - If the state has been expanded into a terminal, try to match this terminal to the current word from sentence. If they match, this parse tree can lead to the result and we continue on the current branch. Otherwise, this state is thrown away, the next tree from the agenda is taken and the main loop is repeated.
- The algorithm finishes once the words from sentence match the leaves of full-expanded parse tree (successful parse), the parse was unsuccessful otherwise.

Two possible approaches can be used for expanding the current tree. First, all possible expansion can be produced in parallel – this strategy is called *breadth-first search* and since the state tree can grow very rapidly at each level, this approach leads to unrealistic amount of used memory.

A more reasonable approach is to use a *depth-first search*. This strategy is described above as the basic top-down parsing algorithm. In figure 3.1 we can see a top-down depth-first derivation for grammar from table 3.1. For the sentence *"This flight includes meal."* it would probably lead to an successful parse.

### 3.2.2 Example of expanding trees

The algorithm starts by assuming that the input can be derived from the $S$. The next step is to find the tops of all the trees which have $S$ on the left side of the production rule. Using grammar $G_1$ in table 3.1 and input sentence, e. g. *"Book that flight"*, we get a hierarchy of trees as shown in fig 3.2. There are three rules expanding $S$ in $G_1$ so three parse trees appear on the second level in figure 3.2. None of the parse trees on the third level which have so far been expanded will eventually match the input sentence.

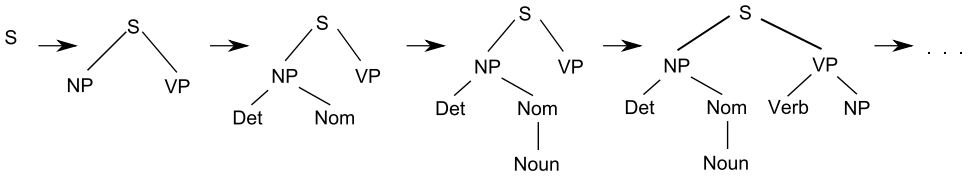


Figure 3.1: A few steps of derivation with top-down depth-first strategy for grammar from table 3.1

| *Production rules* | *Simple lexicon* |
|---|---|
| S → NP VP | Det → that \| this \| a |
| S → Aux NP VP | Nount → book \| flight \| money |
| S → VP | Verb → book \| include \| prefer |
| NP → Det Nominal | Aux → does |
| NP → Proper-Noun | Prep → from \| to \| on |
| Nominal → Noun | Proper-Noun → Houston \| TWA |
| Nominal → Noun Nominal | Nominal → Nominal PP |
| VP → Verb | |
| VP → Verb NP | |

Table 3.1: A miniature English grammar

### 3.2.3 Left-corner parser

The figure 3.1 shows an important qualitative aspect of top-down parsing. Each node expands the leftmost nonterminals along the left edge of the tree. Thus, in any successful parse the current input word must match the first word in the derivation of the unexpanded node which is being processed by the parser. The first terminal along the left edge of the derivation tree is called the *left-corner*.

This brings us a possibility to add a *bottom-up filtering*. A table of all valid left-corner categories can be pre-compiled for each nonterminal, so the constructing and backtracking process for each nonterminal can be avoided. Although it helps to increase the parser efficiency, there are still some problems with that basic top-down parser, as will be discussed below.

### 3.2.4 Problems with the basic top-down parser

#### Left recursion

This definition 6 (on page 5) tells, in other words, that the grammar contains a nonterminal whose derivation includes the nonterminal itself along

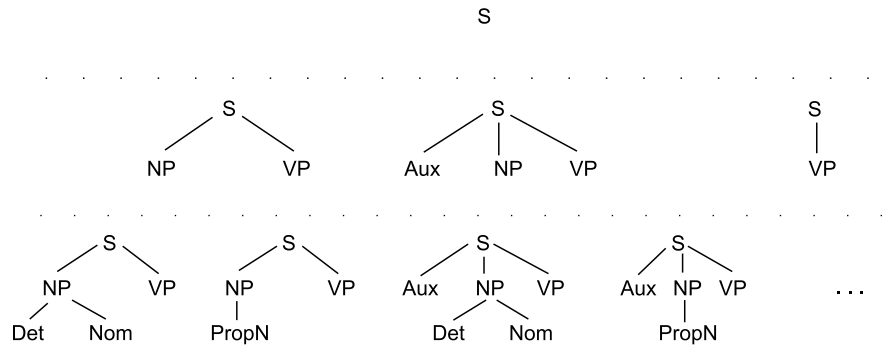


Figure 3.2: Expanding the search space in top-down parsing approach for grammar in table 3.1

its leftmost branch. Consider simple grammar

$$S \rightarrow NP\ VP, \qquad NP \rightarrow NP\ PP, \qquad \ldots$$

Then the depth-first left-to-right parser would produce an infinite tree as shown in figure 3.3.

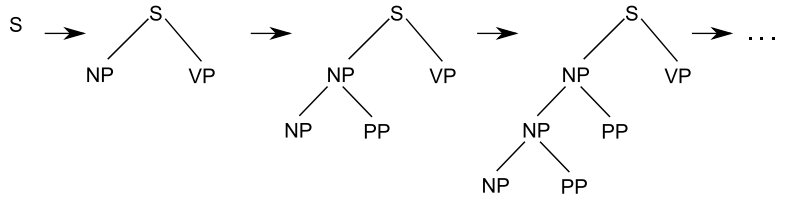


Figure 3.3: The begining of an infinite search caused by rule with left recursion

Two ways of dealing with left recursion are possible. First, we can limit the depth of the backtracking tree. Second, we can rewrite the grammar to an equivalent one (see section 2.2.2) to avoid the left recursion. Unfortunately, rewriting grammar in this way has a major disadvantage – a rewritten phrase may no longer be the most natural grammatical representation of a particular syntactic structure [JM00].

### Ambiguity

The *structural ambiguity* has already been introduced in sec 2.3.2. The ambiguity also plays a very important role in the parsing process.

First, at a certain phase of the parsing process the parser may not have enough information to decide which rule to choose. For instance, there is a

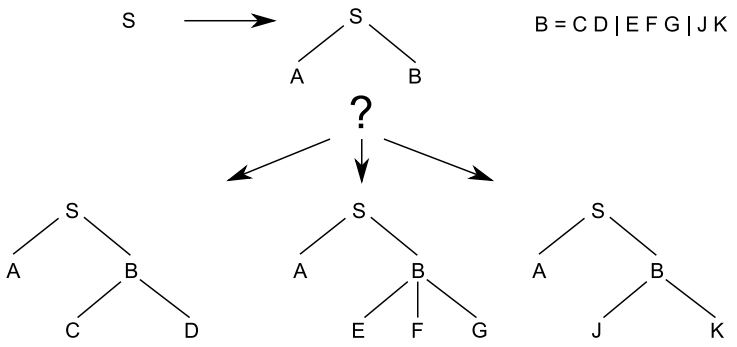


Figure 3.4: Ambiguity in the parsing process

rule $A \rightarrow B$ to be expanded and the grammar contains more rules in form $B \rightarrow \alpha$.

This type of ambiguity complicates the parsing process because the parser either has to expand all the possibilities or it has to try only one rule with the danger that it may later be necessary to backtrack and try another one. An example of this issue for top-down parser is shown in figure 3.4.

Second, it is possible to have more valid parse trees for one input sentence. Again, the parser cannot decide which parse tree really describes the structure. An example of this ambiguity is shown in figure 9.4 (page 54).

*Disambiguation* means that the correct parse tree is chosen. These algorithms generally require both statistical and semantic knowledge. The parser without disambiguation ability simply returns all parse trees for input sentence.

#### Repeated parsing

There is another inefficiency in the basic top down parser. The parser often builds valid sub-trees for parts of input sentence, then discards them during backtracking, only to find that it has to rebuild them again. Except for the topmost component of parse tree, some sub-trees of the final tree can be derived more than once.

The improved version of the top-down parser, which can handle this side-effect of direct backtracking, will be presented in section 3.4.

## 3.3 Basic Bottom-up Parser

While the top-down parsing techniques assume that the input string can be parsed into parse tree with the root containing $S$ (starting symbol), creating

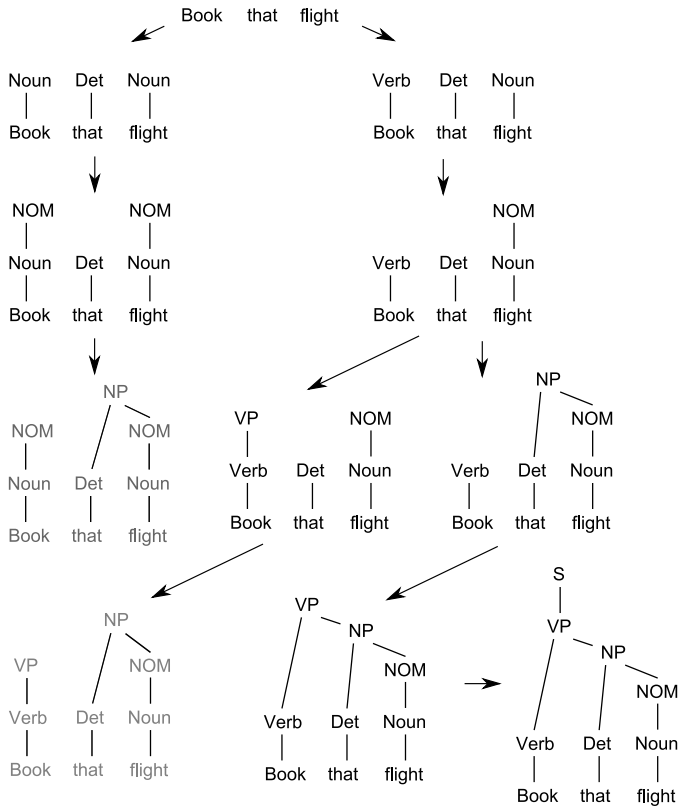


Figure 3.5: Bottom-up search space for the input sentence *Book that flight*. The gray trees represent states which do not lead to the successful parse and, therefore, are pruned from the search space.

a search space by expanding nodes in top-down direction, the *bottom-up* parsers use the opposite approach. The bottom-up algorithm creates the parse tree from terminals, trying to build a wider tree node by concatenating rules that match a production rule from the grammar. In case of successful parsing, the most-top node of parse tree matches the starting symbol $S$ and covers the whole input.

Basic bottom-up parse for the input sentence *Book that flight*, using the grammar from table 3.1 (page 13), is shown in Figure 3.5. The parser must consider two possible sets of trees, since the word *book* is ambiguous; it can be a noun or verb.

### 3.3.1 Algorithm description

The basic algorithm can be formalized as follows:

- Assign terminals to the words from input, looking up in a lexicon.

- Expand search space by creating new trees; this is done by looking for places in the parse-in-progress where the right-hand-side of some rule might fit.
- The parse was successful if there is a parse tree covering the whole input and having starting symbol $S$ as its root. Otherwise, parsing was unsuccessful.

An improved and more efficient version of bottom-up parser will be presented in sections 3.5 and 3.6.

## 3.4 Earley Parser

The Earley algorithm uses a dynamic programming approach to implement a parallel top-down search as discussed in section 3.2. It reduces an apparently exponential-time problem to a polynomial-time problem by eliminating the repetitive solution caused by the backtracking approach. In this case, this algorithm leads to worst-case behavior of $O(N^3)$. For non-ambiguous grammars it can be even $O(N^2)$ [JM00].

**Definition 9** *Given a production rule*

$$S \rightarrow A\ B\ C$$

*where $A$, $B$ and $C$ are terminals or nonterminals, the notation*

$$S \rightarrow A\ \bullet\ B\ C$$

*represents a condition in which $A$ has already been parsed and the sequence $B$ $C$ is expected. This notation is called* the dot notation.

Having an input sentence $x$ consisting of $n$ tokens there are $i$ input positions from 0 to $n-1$. For every input position a *state set* is generated. Each state contains:

- A dot condition for a particular production.
- The origin position. This represents the position at which the matching of this production began.

The state set at input position $i$ is called $S(i)$. The parser begins with the state $S(0)$ which is the top-level rule. Then, the following three stages operate iteratively:

**Predictor.** The job of a predictor is to create new states representing top-down expectations generated during the parse process. It is applied to any state having a nonterminal to the right of the dot that is not a part-of-speech category.

If the state $[A \rightarrow \ldots \bullet B \ldots, j]$ is in set $S_i$, add new state $[B \rightarrow \bullet\alpha, i]$ to $S_i$ for all rules $B \rightarrow \alpha$.

**Scanner.** The scanner examines the input and incorporates a state corresponding to the predicted part-of-speech. Assume that $a$ is the next word from the input, formalized as $a = x_{i+1}$.

If $[A \rightarrow \ldots \bullet a \ldots, j]$ is in $S_i$, add $[A \rightarrow \ldots a \bullet \ldots, j]$ to $S_{i+1}$.

**Completer.** The completer is applied to a state when its dot has reached the right end of the rule. The presence of such a state represents the fact that the parser has successfully discovered a particular grammatical category over some span of the input.

If $[A \rightarrow \ldots \bullet, j]$ is in $S_i$, add $[B \rightarrow \ldots A \bullet \ldots, j]$ to $S_i$ for all states $[B \rightarrow \ldots \bullet A \ldots, k]$ in $S_j$.

All stages are repeated until no more states can be added to the set. It is also important to note that a particular state can be added into set only once.

As for the basic top-down parser, the parse was successful if the final set contains a state in form $[S' \rightarrow S\bullet, 0]$ (we assume the grammar is augmented with a new start rule $S' \rightarrow S$). A good example of using Earley parser can be found in [Ear83] and [JM00].

## 3.5 Cocke-Younger-Kasami Parser

Where the Earley algorithm is essentially a top-down parser which uses a dynamic programming table to efficiently store its intermediate results, the CYK algorithm is essentially a bottom-up parser using the same dynamic programming table. The worst-case time complexity of this parsing strategy is $O(N^3)$ [JM00].

The input of CYK parser is a grammar $G$ in *Chomsky normal form* (CNF, see sec. 2.2.2) and the input sentence constisting of $n$ words $a_1 \ldots a_n$. Although there is a restriction on CNF, any CFG can be processed by this algorithm after transformation to the CNF (as mentioned in 2.2.2).

The output is a table $T$ containing records in form

$$t_{ij}: \quad A \rightarrow a_i \; a_{i+1} \ldots a_{i+j-1},$$

where $A$ is nonterminal from $G$ and $j$ is the length of covered (derived) string.

The algorithm can be described in the three following steps:

**Initialization.** Let $t_{i,1} = A$ for $i = 1, \ldots, n$, where $A \rightarrow a_i$.

**Main loop.** For $j = 2, \ldots, n$ and $i = 1, \ldots, n - j + 1$ repeat:

- We assume $t_{i,k}$ has been set for each $i = 1, \ldots, n - k + 1$ and $k = 1, \ldots, j$. Then we update the table with such

$$t_{i,j} = A,$$

  where $A \rightarrow B\ C$, $k \in \langle 1, j - 1 \rangle$, $B \in t_{i,k}$ and $C \in t_{i+k,j-k}$.
- Since $k < j$ and $j - k < j$, we know that

$$\begin{aligned} B &\rightarrow a_i\ a_{i+1}\ \ldots a_{i+k-1} \\ C &\rightarrow a_{i+k}\ a_{i+k+1}\ \ldots a_{i+j-1} \\ A &\rightarrow a_i\ a_{i+1}\ \ldots a_{i+j-1} \end{aligned}$$

**Accepting parse.** The parse was successful if there is $S \in t_{1,n}$ in the table. It means that the sentence is recognized by the grammar if the substring containing the entire string is matched by the start symbol.

## 3.6 Bottom-up Active Chart Parser

In the previous section, the improved bottom-up parser was discussed. Whereas the CYK parser has a restriction on normal form of the input grammar, there exists a bottom-up parser that can parse any context-free grammar, the *bottom-up chart parser*.

The basic operation in bottom-up parsing is to take a sequence of symbols and match it to the right-hand side of the rules. A simple bottom-up parser could be built by formulating this matching process as a search process. The state would consist of a symbol list, starting with the words in the sentence. Successor states could be generated by exploring all possible ways to rewrite a word by its possible lexical categories and to replace a sequence of symbols that matches the right-hand side of a grammar rule by its left-hand side symbol.

Unfortunately, this simple implementation would be very inefficient, as the parser would try the same matches repeatedly [All95]. Therefore, a data structure called *chart* is introduced to store the partial results of matching done so far, the *edges* (or *arcs*).

### 3.6.1 Active and passive edges

An edge is described by its starting and ending position corresponding to the position of words in the input sentence, which is indexed as shown in figure 3.6.

$_0$The $_1$ large $_2$ can $_3$ can $_4$ hold $_5$ the $_6$ water. $_7$

Figure 3.6: Indexes of input sentences used in edges of the parser

| | |
|---|---|
| S → NP VP | ART → the \| a |
| NP → ART ADJ N | ADJ → large \| big |
| NP → ART N | N → can \| man |
| ⋮ | ⋮ |

Table 3.2: A few rules from simple CFG

It also contains information about the rule that has been (partially) accepted. The rules use the dot notation (see definition 9); the dot preceding a terminal or rule represents an edge where the rules on the left from the dot have already been accepted and the rules on the right from the dot are expected.

Edges in form

$$[i, j, A \rightarrow \alpha_1 \dots \alpha_m \bullet \beta_1 \dots \beta_n],$$

are called *active edges*. $\alpha_{1\dots m}$ and $\beta_{1\dots n}$ are terminals or nonterminals and $\alpha_{1\dots m}$ are optional. Indexes $i$ and $j$ can be equal.

E. g. the following active edge that can be constructed for input sentence from Figure 3.6 and grammar from Table 3.2

$$[0, 3, S \rightarrow NP \bullet VP]$$

tells us that a noun part ($NP$) from index 0 to 3 has been parsed already and the verb part ($VP$) from index 3 is expected. Another active edge describing a hypothesis that there is a $S$ starting at the index 0 can be

$$[0, 0, S \rightarrow \bullet\ NP\ \ VP].$$

In *passive edges* the dot is the last symbol on the right-hand side of the rule. This edge tells that the rule has been parsed completely and covers the part of input sentence between indexes $i$ and $j$.

$$[i, j, A \rightarrow \alpha_1 \dots \alpha_n \bullet]$$

For the input sentence mentioned above, there can be a passive edge

$$[1, 2, Adj \rightarrow \text{large} \bullet]$$

which means there is an adjective between 1 and 2.

### 3.6.2 The fundamental rule

The fundamental rule is used for combining passive and active edges to create new edges. Suppose we have an active edge with the $C$ category immediately to the right of the dot

$$[i, j, \alpha \rightarrow \beta \bullet C\ \gamma]$$

and a passive edge having category $C$ on its left-hand side

$$[j, k, C \rightarrow \delta].$$

These two edges can be combined into the new one covering indexes from $i$ to $k$

$$[i, k, \alpha \rightarrow \beta\ C \bullet \gamma].$$

### 3.6.3 The agenda

The *agenda* is a data structure storing all new edges that have been combined by applying the fundamental rule or crated by hypotheses-making process. Edges are waiting in the agenda until they are added to the chart and further processed by the parsing algorithm.

As with the top-down parser, two search strategies can be used, depending on the agenda implementation. The depth-first approach uses the *stack* implementation, the breadth-first search is implemented as a *queue*.

### 3.6.4 A general algorithm for active chart parsing

The general algorithm of bottom-up active chart parsing can be formalized into the following steps.

1. Initialize *agenda* and *chart*.
2. Repeat until agenda is empty:
   (a) Take first edge $e$ from agenda.
   (b) Add the edge $e$ to the chart (only if the edge is not in the chart already).
   (c) If the edge $e$ is active, use the *fundamental rule* to combine the edge $e$ with all edges from chart. Any edges obtained by applying this rule are added to the agenda.
   (d) If the edge $e$ is passive, make a hypothesis about new constituents based on the edge $e$ and the grammar. More precisely, having the edge

$$[i, j, C \rightarrow \alpha_1 \ldots \alpha_n \bullet]$$

we need to find all production rules starting with $C$ on the right-hand side of the rule. For all these rules

$$\begin{aligned} \alpha &\rightarrow C\ \beta_1 \ldots \beta_n \\ &\vdots \\ \gamma &\rightarrow C\ \delta_1 \ldots \delta_n \end{aligned}$$

we create new active edges starting at position $i$ as

$$\begin{aligned} &[i, i, \alpha \rightarrow \bullet\ C] \\ &\vdots \\ &[i, i, \gamma \rightarrow \bullet\ C] \end{aligned}$$

All newly created edges are put into the agenda.

3. If the chart contains a passive edge from the first sentence index to the last sentence index, the parsing process succeeds. It has failed otherwise.

A complex example of bottom-up chart parsing algorithm can be found in Appendix C.

# Chapter 4

# NLP using the Java programming language

Although the need of using the Java™Programming Language[1] was one of the requirements of the thesis, there are many other reasons why Java can be chosen as the main programming language for the needs of NLP and grammars, respectively. This chapter, therefore, provides an overview of NLP possibilities using Java.

There are also some general advantages of using Java, which can be:

**Platform independency.** This is perhaps the most widely used argument for choosing Java. However, in this case it has a real foundation. NLP application using Java platform can be used on desktop on various operation systems, it can be deployed on an application server and provide a server-side solution, and it can even be used by agents in multiagent systems. Since the JVM[2] is licensed under *Java SE Runtime Environment Binary Code License*, the application can run almost anywhere without any licence restrictions [Sun06].

**Object oriented approach.** This aspect is more focused on developers. The application can be improved (scaled) simply by replacing particular implementations of interfaces with others. Furthermore, using interfaces across the JSAPI makes the application easy-to-test by the possibility of replacing real implementations with its fake implementations (*mock objects*).

There exist "classical" programming languages which are widely used for NLP and computational linguistic, such Prolog, Perl or Lisp. However, the practical part of this thesis is an application which will be integrated into the project LINGVO[3], which is being completely developed in Java. Therefore,

[1] In the rest of the text, simply *Java* will be used

[2] Java Virtual Machine

[3] http://liks.fav.zcu.cz/mediawiki/index.php/LINGVO

the integration costs will be minimal and the application will stay platform independent. It will also simplify application maintenance.

In this chapter, the *Java Speech Grammar Format* (JSGF) will be described first. Although this format is mainly used for speech recognition, algorithms for sentence analysis can be built as well, taking advantage of it. The JSGF will be compared to other grammar formats.

Second, there is the *Java Speech API* (JSAPI) specification, which will be described in the second half of this chapter. This framework[4] provides the core speech technologies for speech recognition and speech synthesis.

## 4.1 Java Speech Grammar Format

Java Speech Grammar Format (JSGF) is a format for textual representation of context-free grammars. The JSGF is platform-independent and vendor-independent and is being used mainly for speech recognition. The specification was developed by Sun Microsystems and submitted to W3C consortium in 2000.[5] Since the new version 2.0 is still in progress in Java Community Process (JCP) as *JSR 113: Java$^{TM}$Speech API 2.0*, the version 1.0 will be introduced.

### 4.1.1 Specification

Only important features of JSGF specification which are used in the practical part of this thesis will be described in this section. Some sections, e. g. *imports*, *weights* etc., will be skipped. For further information see [Sun98].

#### Grammar header

A single file defines a single grammar. The definition of grammar consists of two parts – *grammar header* and *grammar body*. The grammar header contains the self-identifying header and declares the grammar name and list of imports.

The grammar header format is

```
#JSGF V1.0 [char-encoding [locale]];
```

For example a valid grammar header with UTF-8 encoding, which is mostly used in the practical part of this thesis, could be

```
#JSGF V1.0 UTF-8;
```

[4] We assume that the word *framework* is suitable when speaking of JSAPI.

[5] See *http://www.w3.org/Submission/2000/06/* for details.

### Grammar name

Grammar name consist of simple grammar name and, optionally, the package name. The package name convention is similar to package naming in Java. The format is

```
grammar packageName.simpleGrammarName;
```

### Rule definition

The grammar body defines *rules*. Each rule is defined only once and the definition format is

```
public <ruleName> = ruleExpansion;
```

where the `public` keyword is optional. The *rule name* is an unlimited-length sequence of Unicode characters matching the following:

- any character that is legal in an identifier of Java,
- any of following symbols: `+-:;,=|/\()[]@#%!^&~`

The following sections explain the ways in which complex rules can be defined by logical combinations of legal expansions using *composition* (sequences and alternatives), *grouping*, *unary operators* and attachment of application-specific *tags* to expansions.

### Tokens and comments

*Token* is a terminal and is equivalent to single word delimited by whitespace:

```
... token1 token2 "quoted token" token4 ...
```

A *quoted token* is surrounded by quotes and can contain significant whitespaces and some other special characters.

As in Java, JSGF supports two styles of *comments*, a single-line comment (text between `//` and end of line is ignored) and multi-line comment (text between `/*` and `*/` is ignored).

### Sequences

A rule may be defined as a sequence of items which are legal rule expansions. The sequence is a legal expansion itself. An example:

```
<ruleName> = token1 token2 <otherRule1> token3 <otherRule2>;
```

### Alternatives

A *set of alternative* rule expansions separated by `'|'` character has the following form

```
<ruleName> = ruleExp1 | ruleExp2 | ruleExp3;
```

where `ruleExp` are any valid rule expansions (e. g. sequences, tokens, etc.). Alternatives also support *weights* which can be used for the PCFG (see section 2.3.4).

### Grouping

Any legal expansion may be explicitly grouped using matching parentheses `(` and `)`. Grouping has high precedence, therefore the following example

```
<action> = please ( open | close | delete );
```

makes the right sense.

Another grouping is *optional grouping*; square brackets placed around rule expansion indicate that the content is optional. E. g.

```
<ruleName> = [optional sequence] token;
```

### Unary operators

There are three unary operators in JSGF. An unary operator may be attached to any legal rule expansion and it has high precedence.

**Kleene star.** A rule expansion followed by asterisk symbol `*` (Kleene star) indicates that the expansion may repeat *zero or more times*.

**Plus operator.** The rule expansion followed by plus operator `+` may repeat *one or more times*.

**Tags.** Tags provide a mechanism to attach application-specific information to parts of rule definitions. It can be used further for semantic representation in JSGF (e. g. the ECMA scripting language, [Bro07]).

The tag is a string delimited by curly braces `{` and `}`. All characters within the braces, including whitespace, are considered as a part of the tag.

An example combining all three unary operators could be

```
<ruleName> = (zero or many)* (once or many)+
             token {tag content};
```

### 4.1.2 EBNF

EBNF (Extended Backus-Naur form) is an extension of the basic Backus-Naur form (BNF) notation. The EBNF has been standardized by the ISO under the code ISO/IEC 14977:1996(E) [ISO96]. Although there exist more variants of EBNF, we focus only on the standardized version.[6]

#### Difference from JSGF

Both JSGF and EBNF use the same notation for particular expressions. These common notations will be skipped, only the differences between these two formats will be mentioned. A complex example illustrating the differences will be presented later.

**Rule definition.** The rule name is not delimited by angle brackets and can contain whitespaces. As in JSGF, the rule name is followed by equal sign `'='` and ends with semicolon, e. g.

```
rule name = other rule name ;
```

**Tokens.** All tokens are surrounded by quotas, e. g.

```
rule name = "token1" | 'token2';
```

**Sequences.** Items of sequence are separated by *comma*, e. g.

```
rule name = "token1", "token2";
```

**Repetition.** If a symbol can appear *zero or many times*, it is surrounded by curly braces, e. g.

```
rule name = { "optional" } , "nonoptional";
```

If the right curly brace is followed by the minus symbol `'-'`, the symbol can appear *once or many times*, e. g.

```
rule name = { "at least once" } - , "token";
```

For strict $n$-times repetition, the following construction can be used

```
rule = 4 * ("some", "group");
```

which means that the group must be repeated four times.

**Special sequences.** These properties of EBNF have no equal notation in JSGF. EBNF supports *syntactic exceptions* and *special sequences* but their behavior is outside the scope of the ISO standard [ISO96].

[6] A complex overview of various BNF-like notations and differences between them can be found at `http://www.cs.man.ac.uk/~pjj/bnf/ebnf.html`

### 4.1.3 SRGS

The Speech Recognition Grammar Specification (SRGS) defines syntax for representing grammars for use in speech recognition. The syntax of the grammar format is presented in two forms, an Augmented Backus-Naur Form (ABNF) and an XML Form. The specification makes the two representations mappable to allow automatic transformations between the two forms.

**ABNF** is a plain-text grammar representation similar to traditional BNF grammar and to many existing BNF-like representations commonly used in the field of speech recognition including the JSGF from which this specification is derived.

**XML** syntax uses XML elements to represent the grammar constructs and adapts designs from the PipeBeach grammar, TalkML[7] and a research XML variant of the JSGF.

#### ABNF Form

Since the ANBF form is derived from JSGF discussed earlier, only a short overview will be provided.

The rule expansion of a rule definition is delimited at the start and end by the equals sign `'='` and semicolon `';'` respectively. Within rule expansion the following symbols have a syntactic function and delimit plain text.

- Dollar sign `$` and angle brackets `'<'` and `'>'` reference rule
- Parentheses `'('` and `')'` may enclose any rule expansion
- Vertical bar `'|'` delimits alternatives
- Forward slashes `'/'` and `'/'` delimit any weights on alternatives
- Angle brackets `'<'` and `'>'` delimit any repeat operator
- Square brackets `[` and `']'` delimit any optional expansion
- Curly brackets `'{'` and `'}'` delimit any tag
- Exclamation point `'!'` prefixes any language identifier

#### XML Form

The XML form provides the same expression possibilities as the ABNF form. The grammar is stored in XML format and can be validated against DTD or XSD. We will not describe this format here, only an example will be provided in Appendix D. The complete specification can be found in [W3C04].

[7] `http://www.w3.org/Voice/TalkML/`

## 4.2 Java Speech API

The Java Speech API[8] defines a standard cross-platform software interface to speech technology. Two core speech technologies are supported through the JSAPI: *speech recognition* and *speech synthesis*.

JSAPI has been developed by Sun Microsystems in 1998 but it is not a part of JDK[9]. Instead, Sun work with other companies to encourage the availability of multiple implementations.

The specification itself includes only the API documentation for approximately 70 classes and interfaces in JavaDoc format. There are two companion specifications to the JSAPI, the JSGF (see sec. 4.1) and JSML[10]. The specification does not provide any source code or libraries needed to compile application with JSAPI support, therefore the third-party components must be used. This will be discussed in chapter 7.

### 4.2.1 JSGF support in JSAPI

For this thesis, the speech recognition and speech synthesis abilities of JSAPI are not important. The main reason why JSAPI has been chosen is the very good support of JSGF processing in JSAPI.

The specification provides a few basic classes which correspond to the JSGF elements, so any JSGF grammar can be built at runtime. Another very important feature is the ability to parse JSGF grammar from the file and to create the corresponding grammar structure in the memory for further processing.

### 4.2.2 Important classes and interfaces

For grammar processing, there are some important interfaces and classes. They will be briefly introduced in the following paragraphs. All the classes come from package `javax.speech.recognition`.

#### Rule expansion classes

`Rule` is an abstract class which is the ancestor of all the classes describing the rule expansion.

`RuleAlternatives` and `RuleSequence` are used for alternatives (with weights support) and rule sequences.

`RuleCount` and `RuleTag` can be used for unary operators.

`RuleToken` represents a single token (e. g. word).

[8] API = Application Programming Interface
[9] JDK = Java Development Kit
[10] JSML = Java Speech API Markup Language

**Miscellaneous classes**

`RuleGrammar` covers the whole JSGF grammar. It supports run-time rule creation, JSGF grammars parsing, etc.

`Recognizer` is an interface which provides access to speech recognition capabilities, such are grammar management and result handling.

# Part II

# Practical realization

# Chapter 5

# Introduction to Semantic Parsing

The process of extracting the meaning from an utterance by computer and storing the meaning in the computer model is called *Natural Language Understanding* (NLU). *Semantic analysis* is the first step of the NLU process. The goal of semantic analysis is to represent what the subject intended to say.

The problem of *semantic parsing* can be specified by the definition of the input that enters into the semantic parsing algorithm and the output that results from the algorithm. The *input* can be a transcription of the utterance or a word lattice. Prosody may be included as well. The *output* is a context-independent meaning of the utterance in a suitable meaning representation. In short, the result of semantic parsing is required to support the contextual interpretation that follows the process of semantic analysis [Kon06].

## 5.1 Stochastic semantic parsing

Recent trends in the area of NLP are heading towards making all the processes stochastic. In a stochastic method of semantic parsing, semantic knowledge is usually represented by some kind of a tree. Relations between semantic labels and the corresponding words are learned automatically from a large annotated training corpus and stored in the form of model parameters.

### 5.1.1 Hidden Vector State Model

The Hidden Vector-state Markov (HVS) model is described in [HY05]. The HVS model is trained by a partially unsupervised learning method. This model is able to describe the sentence structure and relation between words [Kon06]. This model has been chosen for the LINGVO project.

*Identification of lexical classes* is the fundamental step of semantic analysis. Having obtained a set of lexical classes, a tree is then built upon it. This tree describes the relation between lexical classes and their appropriate superior concept (e. g. lexical class 'Time' belongs to the concept 'Date/time of departure' and this concept belongs to the 'City Bus' superior concept).

### Model training

There are two basic options for stochastic model training – *supervised* and *unsupervised* training. Fully unsupervised training is impossible for a non-trivial domain, therefore a partially unsupervised training is used. A large set of annotated sentences is required for the model training.

*Annotated data* are the data that are augmented with some kind of additional information. For example, during semantic annotation, words from a sentence are associated with their meaning representation. *Abstract annotation tree* is then a structure used for semantic description in NLU [HY05]. The annotations are mostly created by human annotators. This process is very expensive.

In practice, provision of a fully annotated training set is not realistic. Training from partially annotated corpora is, therefore, crucial for practical applications. Consequently, prior knowledge of the domain is required. This knowledge consists of the two following parts:

- A set of domain specific and/or generic lexical classes.
- Abstract semantic annotation for each utterance.

*Lexical class* is a word or word group with a similar specific meaning. For instance, in a flight information system, a typical class might be CITY = {'London', 'San Francisco'}. There might also be generic lexical classes which are independent of the domain, e. g. time and date. Given these classes, all words in the sentence are replaced with corresponding lexical class when possible. This reduces the size of the model.

The *abstract semantic annotation* describes the relationship between elements and lexical classes in the sentence. It is not necessary to annotate each word from the sentence; words with semantic meaning should be considered particularly. An example of semantic annotation is shown in figure 5.1.

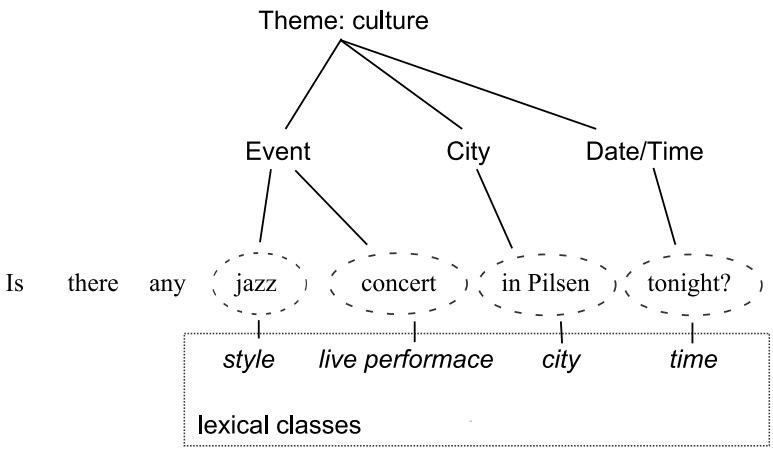


Figure 5.1: An example of abstract semantic annotation tree

# Chapter 6

# Java Abstract Annotation Editor

This chapter describes a graphical editor that was designed for the creation of annotated data for training stochastic models. *Abstract annotation trees* and *lexical classes* have been introduced in chapter 5. The main goal was to develop a flexible editor for the LINGVO project.[1]

The Java Abstract Annotation Editor (JAAE) is an graphical editor intended for sentence annotation and annotation tree building. Since the annotation process is highly time-consuming and expensive, a very efficient and user friendly editor is required.

## 6.1 Editor features

The JAAE is a standalone desktop application written in Java. It was originally designed for annotation of sentences by Abstract Annotation methodology (see [HY05]). All the data files are stored in XML format.

An annotation tree must be built upon a predefined structure, the *annotation schema*. It guarantees the tree validity; that means that each element can appear only at an allowed position. More formally, the annotation tree is a sub-tree of the annotation schema, where the root is a theme of sentence and the leaves are lexical classes (but not necessary). The annotation schema is very domain specific and should be created after domain research.

There are two input files and one output file. The first input file is a set of sentences. The second one is an annotation schema. The output file stores the sentences and their annotation tree.

[1] The LINGVO project is one of the research projects at the Laboratory of Intelligent Communication Systems. This laboratory is situated at the Department of Computer Science and Engineering, Faculty of Applied Sciences, University of West Bohemia. 'Lingvo' is a word for 'language' in Esperanto. This project focuses on recognition, syntactic and semantic analysis and interpretation of spoken sentences.

A XML file with a set of sentences has a very simple structure. The root XML element is `<dialog>`. The text of sentences is placed within the elements `<request>`. All other elements and attributes of elements are ignored so the input file can carry additional information (for example, system responses in a dialog transcription). So, in the following example the elements `<response>` and `<event>` will be ignored by the editor.

```
<?xml version="1.0" encoding="utf-8"?>
<dialog>
    <request>Text of the sentence 1.</request>
    <response>Optional text of a system response.</response>
    <request>Text of the sentence 2.</request>
    <event>Additional optional information</event>
    ...
</dialog>
```

The annotation scheme XML file and the output document XML file are slightly more complicated. The structure of these files is described in the XSD files.

The main features of the editor are as follows:

**Building trees and assigning lexical classes.** Users can choose the main theme of the source sentence, build the abstract semantic tree and assign the lexical classes to any word combination.

**User interface.** The GUI is user-friendly and the application has native look and feel, as shown in figure 6.1.

**Custom annotation schemas.** Users can create their own annotation schemas in XML format according to XSD validation schema. Custom schemas can be loaded manually or automatically during the application start.

**Problematic phrases.** Any part of annotation tree including lexical classes can be marked as problematic with an appropriate reason.

**Internationalization.** The application supports localized messages; translations can be added easily by editing XML file. Currently, the English and Czech versions are included.

**Auto-save.** The current document is periodically saved to temporary file.

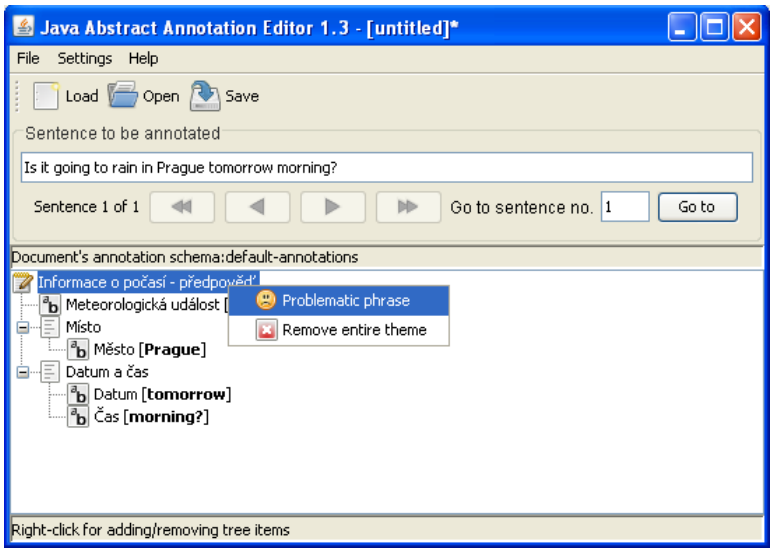


Figure 6.1: JAAE graphical user interface

## 6.2 Implementation

The application has been written in Java and JFC Swing. Although J2EE provides access to processing XML files, 3rd part library has been chosen (as described below). For XML document validation, XML Schema instead of DTD is used.

### 6.2.1 XOM library

The XOM Java library is used for processing XML files. It is an open source tree-based API for processing XML. XOM has been developed as a correct, simple and performance XML library, in comparison with standard Java XML API[2]. The main advantages of this library are:

- *memory efficient;* allows to filter document as it is built and skip uninteresting nodes immediately, can process documents greater than 1 GB
- *dual streaming/tree-based API;* individual nodes of the tree can be processed while the document is still being built.
- *supports other XML technologies;* for instance namespaces, XPath, XSLT, XInclude, etc.

[2]At `http://www.xom.nu/whatswrong/` there can be found more information about disadvantages of XML API

### 6.2.2 XML Schema

For validating the annotation schemas the XML Schema validation has been chosen. The XML Schema language has been published by W3C and it can be used to express a schema: a set of rules to which an XML document must conform in order to be considered 'valid' according to that schema. XML Schema allows the definition of elements and attributes which can appear in the document, relations between elements, element content and element or attribute data-types as well.

More information about XML Schema can be found at W3C website `http://www.w3.org/XML/Schema`.

### 6.2.3 Annotation schema model

The model of annotation schema is shown in figure 6.2. The annotation schema has a tree-structure, where all branches and leaves are inherited from abstract class `AnnotationModelNode`. There is also binding between `AnnotationModelLexicalClass` and `LexicalClass` because lexical classes are unique and shared across the whole annotation schema.

The singleton class `LexicalClassManager` manages all lexical classes and loads them from XML. There is also a factory class `AnnotationModelNodeFactory` which contains static factory method for creating instances of `AnnotationModelNode` according to the method parameter.

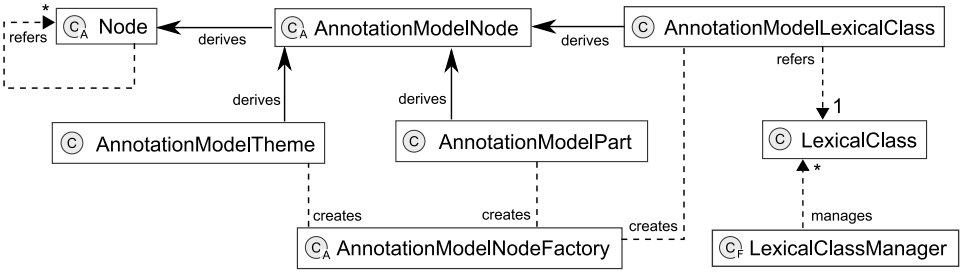


Figure 6.2: UML class diagram of annotation schema model

### 6.2.4 Annotation document model

The editor works with documents containing one or more sentences with the corresponding abstract annotation tree. This model is implemented as shown in figure 6.3.

The main class is `AnnotationDocument` which aggregates one or more instances of `SentenceAnnotation`. This class represents a single editable object – a sentence to be annotated (`Sentence`) and appropriate root of annotation tree, an instance of `AnnotationModelTheme`.

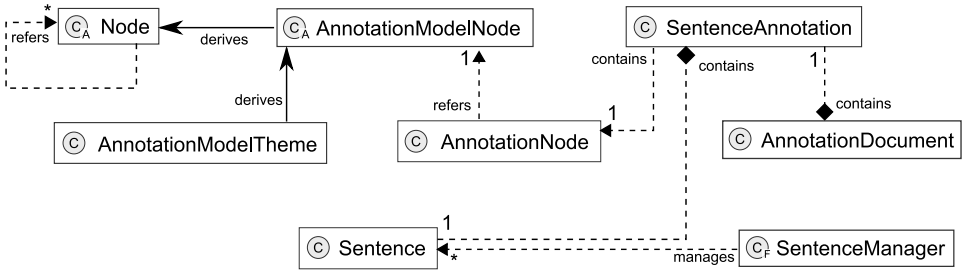


Figure 6.3: UML class diagram of document

The annotation tree consists of `AnnotationNode` objects. Each object is bound with an appropriate node from the annotation model, as shown in figure 6.4.

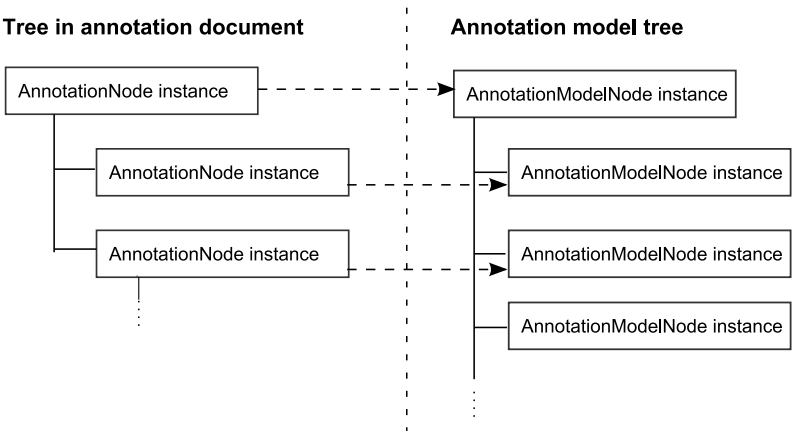


Figure 6.4: A binding between annotation document and annotation model

### 6.2.5 Graphical user interface

The graphical user interface is built with JFC/Swing. The main goal was to separate application logic and the view layer. The application does not use the MVC model fairly. However, there is the business logic (`AnnotationDocument` instance), the view logic (`MainForm` instance) and objects inherited from `Action` class. Actions are bound to GUI controls and perform document changes.

### 6.2.6 Building

The JAAE editor must be compiled with Ant and Java SDK 1.5 or higher. Ant is a Java tool for automatic building and deploying java applications

and can be obtained from `http://ant.apache.org`. The following tasks can be used for application development:

- `ant compile` – compiles the whole project
- `ant dist` – creates a single `.jar` file in the `dist` directory; this file is designated for application distribution to the end users
- `ant clean` – removes the `dist` and `bin` directories and the application log as well

# Chapter 7

# Java Speech API implementation

The JSAPI specification has already been introduced in section 4.2. The practical part of the thesis uses this framework for processing JSGF grammars. The parsers are built using a part of JSAPI as well.

In this chapter, the implementation of JSAPI will be discussed. The standard itself has already been described, so we focus on the existing implementations. Although Sun Microsystems does not provide any libraries or source code of JSAPI, there exist projects which use their own JSAPI libraries. They will be introduced in the first part of this chapter.

In the second part, our own implementation of JSAPI will be described. It does not cover the whole framework, only the parts handling JSGF grammars are implemented. Despite that, it is worth mentioning.

## 7.1 Existing implementations

When choosing an appropriate implementation, the most important attribute is the licence. All the programs which are product parts of this thesis are licenced under BSD-like licence. This allows users to modify it, redistribute it and include into other projects freely. Thus, the chosen JSAPI implementation had to be licenced under BSD-like licence as well.

In the FAQ of JSAPI specification there is a list pointing to existing implementations of JSAPI. There are some of them:

**FreeTTS** The *FreeTTS* project is a speech synthesizer written entirely in Java and is licenced under BSD Licence. But it does not provide full support for the JSAPI, only the subset `javax.speech.synthesis` is supported. The latest version comes from 2005.

**IBM Speech for Java** This project has retired.

**TalkingJava SDK with JSAPI** This product from CloudGarden has produced a full implementation JSAPI for Windows platforms. The product is closed-source and non-free. The latest version is from 2004.

**CMU Sphinx** Sphinx is a speaker-independent large vocabulary continuous speech recognizer released under Berkeley's style license. It is also a collection of open source tools and resources that allows researchers and developers to build speech recognition systems.

Sphinx contains a JSAPI implementation, but it is not available under BSD licence. Instead, the binaries are available under a separate binary code licence.

The result of this survey is interesting – there is no open-source implementation of JSAPI. Therefore, we wrote our own implementation of the JSAPI part for grammars processing.

## 7.2 Implementation of basic classes

The JSAPI is divided into three packages. The common package for entire JSAPI is `javax.speech`. The package containing recognizer's classes is `javax.speech.recognition`. The third one is `javax.speech.synthesis`.

The specification does not declare interfaces and abstract classes separately, interfaces and implementing classes are mixed in the same package.[1] E. g. the class `RuleToken` is in the same package as the interface `RuleGrammar`. Therefore the original naming convention of packages was retained, instead of moving the implementation into e. g. `cz.habi.jsapi.impl`.

### 7.2.1 The `javax.speech` package

From the `javax.speech` package we had to implement approximately 20 classes and interfaces. They do not provide any functionality in relation to grammar processing, but they are essential for the whole JSAPI framework functionality.

This package contains various exception classes (e. g. `EngineException`, `SpeechException`), events (e. g. `AudioEvent`, `SpeechEvent`) and interfaces with definitions of constants. The integer values of these constant have not been chosen randomly; since the constants are defined as `public` in Sphinx JSAPI library, the same values are used.

[1] Usually, if there exists a specification to be implemented, it is described by a set of interfaces. Then the implementation is created separately, thus the interfaces and the implementation are independent. This design pattern is called *bridge* [GHJV95]

### 7.2.2 The `javax.speech.recognition` package

The `javax.speech.recognition` package contains approximately 35 classes and interfaces, some of them are important for handling rules and grammars. Not all of them are implemented completely. The development was primarily focused on classes with relation to `RuleGrammar` interface and classes implementing rule expansion.

Important classes from this package have been introduced in section 4.2.2 already. Other essential classes are implemented partially, therefore some methods can throw a runtime exception `NotYetImplementedException`. However, these methods have no relation to grammar processing.

### 7.2.3 The `javax.speech.recognition.impl` package

Although the `javax.speech.recognition` package contains interfaces and implemented classes mixed, there are some interfaces that can be implemented in separated package. They are in the `javax.speech.recognition.impl` package. The most important class is `RuleGrammarImpl` which is almost fully implemented `RuleGrammar` interface. As introduced in 4.2.2, this class provides a complex grammar functionality.

In a general JSAPI-based applications, the handling of JSGF grammars is only a part of the their application scope. They would probably use a recognizer or a synthetizer. For recognition purposes, there is the `Recognizer` interface. Its implementation should provide a full control over handling audio resources, speaker and vocabulary management etc. The implementation of such a class would be very exhausting and its functionality is out of the scope of our needs. Therefore, a simple skeleton `RecognizerSkeletonImpl` was created. Its body is almost empty, but this class is necessary for JSGF parser.

Having implemented these classes, we should be able to create and process grammars by a program. However, a very important feature of JSAPI is to load grammars from JSGF files. This will be discussed in the following section.

## 7.3 Loading and parsing JSGF grammars

Writing the JSGF grammar parser from the scratch would be very time consuming. Since there already exists a parser in Sphinx project, its modification was used in our JSAPI implementation. It is based on JavaCC – the *Java Compiler Compiler*.

### 7.3.1 JavaCC

JavaCC is a generator of parsers and lexical analyzers that is licenced under BSD licence. It is similar to Yacc[2] because it generates a parser for a grammar provided in EBNF notation. The output of JavaCC is Java source code that implements a top-down parser. The parser is limited to the *LL(k)*[3] class of grammars.

#### Input and output files

The source file `jsgf.jj` is in the `javax.speech.recognition.impl.parser` package. After processing with `javacc` these Java source files (public classes) are produced:

- `JSGFParser` – the JSGF parser class,
- `JSGFParserConstant` – an interface associating token classes with symbolic names,
- `JSGFParserTokenManager` – the token manager,
- `ParseException` – exception indicating that the input did not conform to the input grammar,
- `SimpleCharStream` – representing the stream of input characters;
- `Token` – a single input token passed to the parser;
- `TokenMgrError` – an error thrown from the `TokenManager`.

The input file can be divided into three sections – the *tokens declaration* (lexical analyzer), *semantic actions* (specifying the parser) and the *parser class body*.

#### Lexical analyzer

The *tokens* are defined by a *regular expression*. E. g. numbers can be defined as

```
TOKEN : { < NUMBER : ( ["0"-"9"] )+ > }
```

or definition of reserved words in JSGF is

```
TOKEN : {
    < GRAMMAR: "grammar" > |
    < IMPORT: "import" > |
    < PUBLIC: "public" >
}
```

[2] Yacc = Yet Another Compiler Compiler; a parser generator on many Unix systems

[3] *LL(k)* parsers process the input from left to right and construct the leftmost derivation of the sentence; see section 3.2

### Specifying the parser

The methods of parser are specified as follows:

```
ResultClass MethodName() :
{ local variables }
{ regular expression of tokens or other methods }
```

The following example shows a piece of code, where reference to another rule within the rule expansion is declared.

```
1: RuleName ruleRef() :
2:   { String s; }
3:   {
4:     ("<" s = Name() ">")
5:     {
6:       RuleName rn = new RuleName(s);
7:       return rn;
8:     }
9:  }
```

The method of this parser returns an instance of `RuleName`. At line 2 there is a local variable `s` which is then returned by the method of the other parser `Name()` (line 4). Between the lines 5 and 8 there is a Java code which handles the situation when the input tokens from line 4 were accepted.

### The Parser body

The parser body is similar to Java class definition. It has the following form:

```
options {
    ... some global JavaCC options
}

PARSER_BEGIN(JSGFParser)
   package javax.speech. ...

   imports...

   public class JSGFParser {
      class body...
   }
PARSER_END(JSGFParser)
```

The `JSGFParser` class contains only static methods for creating rules from JSGF source. The global option `JAVA_UNICODE_ESCAPE` is set to false because we accept input file in any encoding.

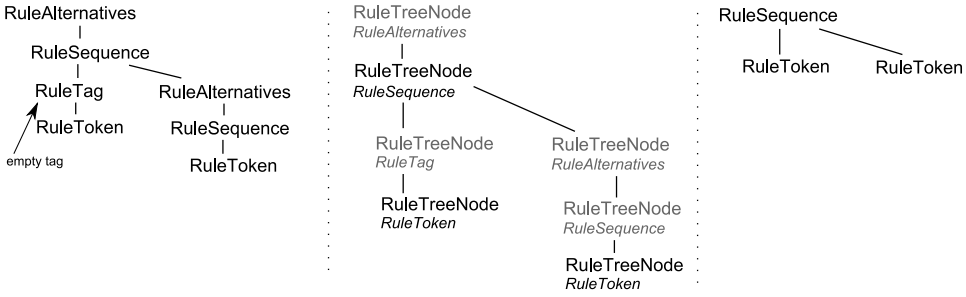


Figure 7.1: Optimization of rules after loading JSGF grammar. On the left there is an original parsed structure (containing empty tag and single-subchild alternatives and sequences). In the middle there is an appropriate tree structure consisting of `RuleTreeNode` instances. After transforming back (on the right), the gray nodes are pruned away.

### 7.3.2 Rules optimization

After parsing grammar from JSGF and creating the appropriate grammar structure in memory, the phase of *optimization* follows.

There is one very important class which is not a part of original JSAPI, the `RuleTreeNode`. Although the JSAPI specifies the whole rule expansion with classes derived from the `Rule` abstract class, the tree structure of such expanding is not apparent. Therefore, a node class containing a `Rule` instance was created.

This class has reference to its parent node and it contains a sorted list of sub-nodes. Therefore, any rule expansion can be transformed into tree structure consisting of `RuleTreeNode` and backwards, respectively. This transformation is used for rules optimization, the tree structure of rule expansion is also used by bottom-up parser (will be described in chapter 8).

Once the grammar is loaded, all rules are transformed into a tree structure. When transforming back to the JSAPI representation, some optimization is performed. E. g. rule sequences which contain single sub-child or which are empty, are skipped. Empty tags are pruned away as well. An example of such optimization is shown in Figure 7.1.

## 7.4 Building

The *Ant* tool is used for code generating, compiling, deploying and testing the whole project. The compilation is divided into few steps, each with corresponding ant task.

- `ant generate-src` – This task runs the JavaCC and creates a parser from source `*.jj` file. The generated Java source files (see section 7.3.1

for detail list) are stored in `build/src-gen/`.

- `ant compile` – This task compiles the project using generated sources from `build/src-gen/` and sources from `src/`. Output classes are in `build/bin/`.
- `ant dist` – Task for deploying JSAPI framework. It is the default task, also running the compilation. The library `jsapi-1.0.jar` is created in `build/dist/`.

There are some additional tasks, such `clean` and `javadoc` with standard functionality.

## 7.5 Unit testing

A very helpful tool while developing the JSAPI implementation was the *unit testing*. Since there is a quite strict specification of each method's behavior, writing the test cases guarantees the proper functionality across various inputs and object states.

The tests are written using *JUnit* library[4] and can be found in `tests/Java/src/`. They cover all important classes from JSAPI with relation to grammar processing.

The tests can be compiled using *Ant*, by running task `compile-tests`. Tests can be performed by task `tests` or using an IDE[5], such Eclipse etc. Having all tests "green" means that all tests passed successfully.[6]

## 7.6 Limitations and known issues

The implementation of the specification is not complete yet. There are some well-known issues which are not crucial for this thesis.

- Imports are not yet supported. The basic background code for import is written basic background code for imports is written (parsing, etc.) but resolving rules from other grammars is not implemented.
- Support for `<NULL>` and `<VOID>` is not implemented. These are JSGF specific rules designed for testing purposes.

[4] http://www.junit.org

[5] IDE = Integrated Development Environment

[6] Unfortunately, it does not mean that the software is bug-free. It only means that we need more tests [Bac02].

# Chapter 8

# Bottom-up Chart Parser Implementation

The algorithm of bottom-up active chart parser has already been described in section 3.6. A short reminder: this algorithm reads input words and attempts to build syntactic trees from the bottom to the top. It uses the *chart* for storing results and the *agenda* for newly created *edges*. Parsing has been successful if there is an edge containing the grammar's starting symbol that covers the whole input.

This chart parser implementation is based on JSAPI implementation introduced in the previous chapter and it is also written in Java. The reason, why our own implementation of JSAPI is used instead of using some third party libraries, is that a tree structure (described in 7.2) is further used by the parser. The JSAPI specification does not provide any methods for building parsers upon it, so the `RuleTreeNode` wrapper is widely used by this bottom-up parser.

In this chapter the implementation of bottom-up chart parser will be introduced. Some differences between the "standard" parser and this one will be mentioned. We will also focus on the *dot* moving, the parse tree pruning and ambiguity.

## 8.1 The dot movement

The *fundamental rule* is used by parser for creating new edges by combining two existing ones (see sec. 3.6.2). For simple rule expansions, e. g. sequences, the *dot* has only one possible follower.[1] But what if the rule expansion is more complex and the dot is followed by an optional rule or even rule alternatives? It brings a little difficulty to the implementation.

[1]The *rule category following the dot immediately* will be simply called *follower*.

| Production rule | Possible followers of the dot |
|---|---|
| $\alpha \bullet A\ \beta$ | $A$ |
| $\alpha \bullet (\ A \mid B\ )\ \beta$ | $A$, $B$ |
| $\alpha \bullet [\ A\ ]\ B\ \beta$ | $A$, $B$ |
| $\alpha \bullet A^*\ B\ \beta$ | $A$, $B$ |
| $\alpha \bullet A^+\ B\ \beta$ | $A$ |
| $\alpha\ (\ \beta \mid \gamma\ \bullet\ [\ A\ ]\ )\ [\ B\ ](\ [\ C\ ] \mid D\ )$ | $A$, $B$, $C$, $D$ |

Table 8.1: Possible followers of the dot for various rules.

| First production rule: $[i, j, S \rightarrow \alpha\ (\ \beta \mid \gamma\ \bullet\ [\ A\ ]\ )\ [\ B\ ](\ [\ C\ ] \mid D\ )\ ]$ | |
|---|---|
| Other rules | Result |
| $[j, k, A \rightarrow \delta]$ | $[i, k, S \rightarrow \alpha\ (\ \beta \mid \gamma\ [\ A\ ]\ ) \bullet\ [\ B\ ](\ [\ C\ ] \mid D\ )\ ]$ |
| $[j, k, B \rightarrow \delta[$ | $[i, k, S \rightarrow \alpha\ (\ \beta \mid \gamma\ [\ A\ ]\ )\ [\ B\ ]\ \bullet\ (\ [\ C\ ] \mid D\ )\ ]$ |
| $[j, k, C \rightarrow \delta[$ | $[i, k, S \rightarrow \alpha\ (\ \beta \mid \gamma\ [\ A\ ]\ )\ [\ B\ ]\ (\ [\ C\ ] \mid D\ )\ \bullet]$ |
| $[j, k, D \rightarrow \delta[$ | $[i, k, S \rightarrow \alpha\ (\ \beta \mid \gamma\ [\ A\ ]\ )\ [\ B\ ]\ (\ [\ C\ ] \mid D\ )\ \bullet]$ |

Table 8.2: An example of applying the fundamental rule to more complex rule expansion.

### 8.1.1 Possible followers

Thus, we need to get a list of all *possible followers* to any rule category from the rule expansion. This list tells us that any rule from that list is an acceptable rule. Table 8.1 shows some examples of rules following the dot.

When combining such complex rule expansions, all possible followers must be considered by the fundamental rule. E. g. having the last rule expansion from table 8.1, results of applying the fundamental rule are shown in table 8.2.

## 8.2 Building parse tree

If the parsing was successful, the chart contains an edge covering the whole input. Such edge tells us only that the parser succeed. It has no information about the syntactic structure of the input (see section 2.3.2). Thus, the parse tree must be created during the parsing.

Each edge contain information about its ancestors. These are the edges that the current edge was created from. E. g. when combining edges

$$[0, 1, S \rightarrow \langle \text{TENS} \rangle \bullet \langle \text{NUMS} \rangle] \qquad \text{and} \qquad [1, 2, \text{NUMS} \rightarrow \text{one} \bullet],$$

the newly created edge

$$[0, 2, S \rightarrow \langle \text{TENS} \rangle \langle \text{NUMS} \rangle \bullet]$$

has both edges as ancestors. They correspond to the children in the syntactic tree. In the final parse tree only the passive edges are kept.

## 8.3 Implementation

### 8.3.1 Class diagram

The object oriented design is simple and the model follows the algorithm description. The class diagram is shown in figure 8.1. The following important classes are:

`Edge` – an edge description, including starting and ending position, left rule name and the rule expansion. It also provides methods for combining edges.

`Agenda` – an abstract class. It has two implementations – the `StackAgenda` (uses a stack for storing edges, LIFO) and the `QueueAgenda` (uses a queue for storing edges, FIFO).

`Chart` – simple class for storing edges, the Chart. After parsing, the Chart returns a list of successfully parsed trees (the list may be empty).

`ChartParser` – the main class implementing the parsing algorithm. The parser's consturctor requires a `RuleGrammarImpl` grammar, the starting symbol of the grammar, the input sentence (it is represented by `TokenProducer` class) and the type of the agenda. The agenda is then initialized by using refection.[2]

### 8.3.2 Building and testing

Again, the *Ant* is used for compiling and building the application. Since the chart parser is not standalone application, the testing and building will be described in the following chapter.

[2]The constructor of `ChartParser` has parameter `agendaClass` of type `Class<? extends Agenda>`. Then the instance is created by calling `agendaClass.newInstance();`.

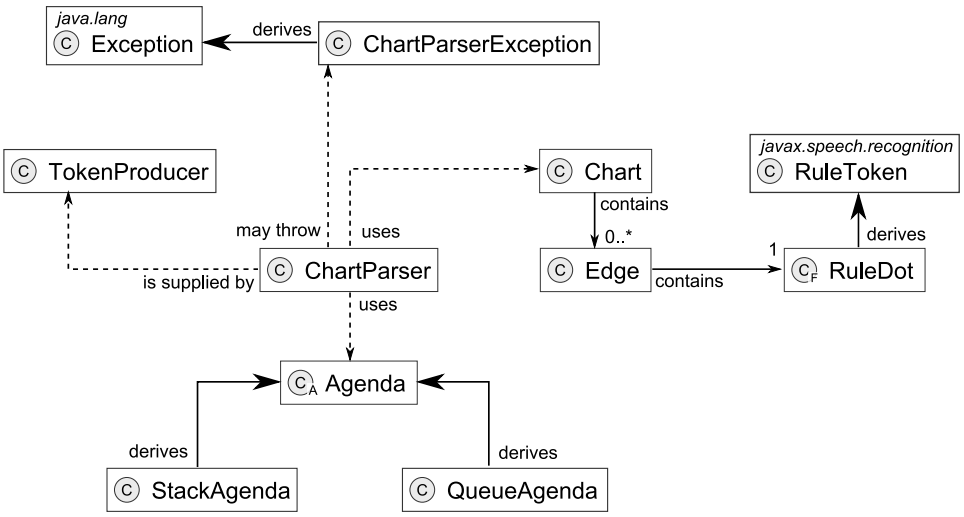


Figure 8.1: Class diagram of the chart parser implementation.

# Chapter 9

# Lexical Class Analysis

The main task of this thesis was to explore and develop a set of appropriate methods for lexical class identification in natural language sentences. *Lexical classes* have been introduced in chapter 5 — *lexical class* is a single word or a word group with specific semantic information, such dates, times, cities etc.

This chapter describes an implementation of lexical class identification based on context-free grammars and parsing methods. The main algorithm will be described together with the experimental results.

## 9.1 Lexical class identification

The lexical classes usually have a consistent structure which can be described by context-free grammars. Having obtained a set of annotated sentences with assigned lexical classes (e. g. date and time) created by human annotators, we can develop an appropriate grammar for describing certain lexical classes.

For the identification process, a *local parsing method* can be used. Chapter 3 introduced various parsing algorithms with different parsing approaches. The top-down parsing methods are inappropriate for lexical class identification because they assume that the starting symbol covers the whole input sentence. Since lexical classes are structures consisting of a few words, a bottom-up parser seems to be an efficient tool for this identification.

In this thesis, the bottom-up active chart parser is used (see section 3.6). It parses a context-free grammar without restriction to any normal form. Moreover, it is still an efficient parsing method [All95].

### 9.1.1 Preprocessing

In the initialization part, all the input grammars are loaded. The grammars are hand-written and they are stored in text files in JSGF format (see section

4.1). Single grammar describes a single lexical class, hence the parser must gain information about the name of the lexical class and the starting symbol of the grammar.

The input sentence is split into words (tokens). The parser looks up in the grammar whether the token is present on the right side of any rule. Tokens which are not a part of the grammar are skipped. This preprocessing phase splits the input sentence into some clusters as shown in figure 9.1.

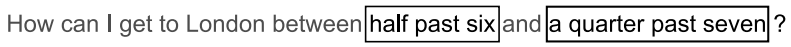


Figure 9.1: The input sentence contains two possible lexical classes with date/time information. The tokenizer feeds the parser with two groups of words.

### 9.1.2 Parsing

Having obtained the input string, the parsing process can begin. The parser attempts to build the syntactic tree over the input words (see section 3.6 for details). The parser succeeds if there is a syntactic tree with starting symbol as the root node. Figure 9.2 shows such a result for grammar describing date and time; the grammar used would be very simple.

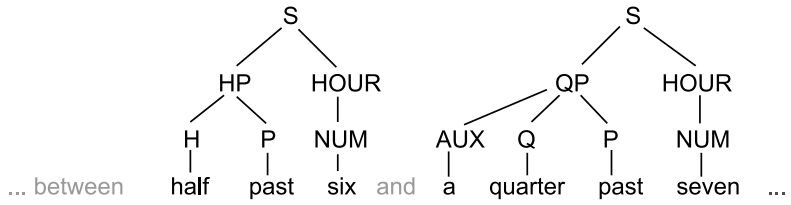


Figure 9.2: Successfully parsed lexical classes from figure 9.1.

Let us have the following input sentence in Czech: *Dvě stě dvacet pět šedesát šest banánu.* (Two hundred and twenty-five sixty-six bananas) and the grammar generating numbers. The phrase makes no sense, but it contains two grammatically correct consecutive numbers – 225 and 60. From another point of view, there can be numbers 200, 20, 5, 60 and 6 (in Czech only). These possibilities are shown in figure 9.3.

Thus, the parser chooses *all the widest trees from the left that have the starting symbol as the root.* In our case, the tree covering the first four words (the number 225) and the tree covering two remaining words (the number 66) are chosen. The parsing was successful, according to our expectation.

Ambiguity is a major issue in parsing. Let us suppose we have an ambiguous grammar for a certain lexical class. If there are two or more trees

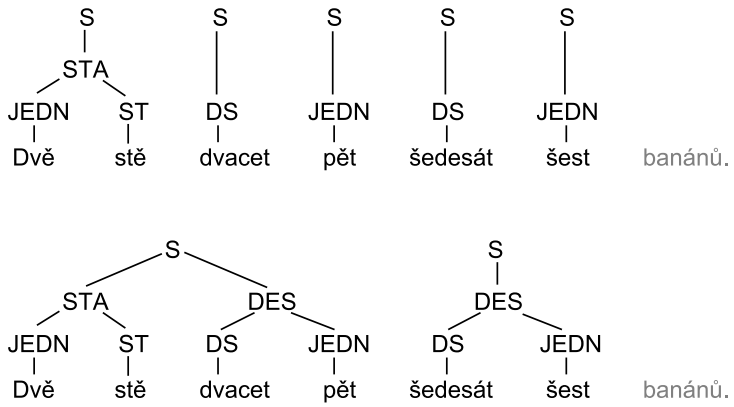


Figure 9.3: There are more combinations of possible successful parse trees, as shown above. The parser chooses the two widest trees.

from position $i$ to $j$ with the starting symbol as root (as shown in figure 9.4), the parser cannot decide which tree is the right one without any additional information (probabilities, etc.). However, this is not an issue in the process of identification. We simply say that there is a lexical class between indexes $i$ and $j$.

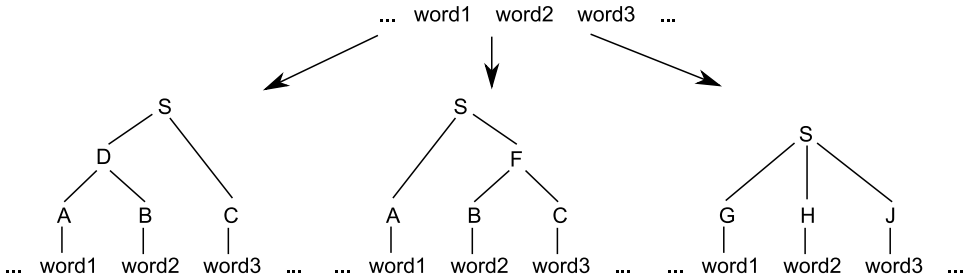


Figure 9.4: Three different successful parse trees over the same input.

Another problem can be an overlap of parsing trees of two different grammars. It means that there are two successful parse trees $T_1$ and $T_2$, where $T_1$ is generated by grammar $G_1$ and $T_2$ is generated by grammar $G_2$. The tree $T_1$ if from index $a$ to $b$, the tree $T_2$ from $x$ to $y$ and $x \leq b$. Again, in this case we cannot decide which lexical class is really present in the sentence and which lexical class has been identified as wrong. This problem is illustrated in figure 9.5.

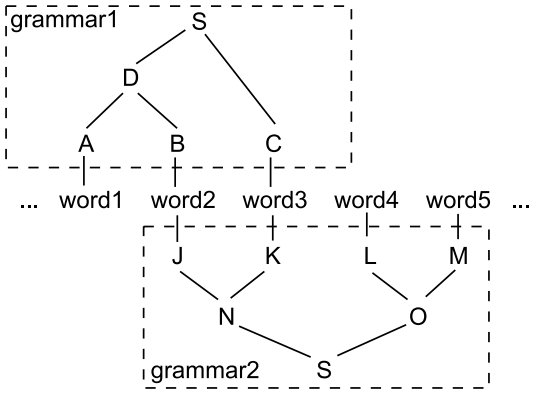


Figure 9.5: Two overlapping parse trees for two grammars describing different lexical classes.

## 9.2 Input and output

All input and output files are in the XML format. The format of the input is the same as the format of input file for JAAE (see page 36). The file contains a list of user queries in which the lexical classes are to be identified.

The output file is another XML file with the following structure (the input sentence is *'Is there any bus number twenty-two from Bory around ten o'clock tomorrow morning?'*):

```
<?xml version="1.0" encoding="utf-8"?>
<sentences>
    <sentence source="Jede mi zejtra kolem deseti dopoledne
        autobus dvacet dva z Bor? ">
        <lexicalClass class="time" from="3" to="6">
            kolem deseti dopoledne
        </lexicalClass>
        <lexicalClass class="date" from="2" to="3">
            zejtra
        </lexicalClass>
        <lexicalClass class="number" from="7" to="9">
            dvacet dva
        </lexicalClass>
    </sentence>
    ...
</sentences>
```

The program applies all registered grammars to each input sentence. In the example above, the input sentence contains three lexical classes, each of different type. The source sentence appears in the output as the `source`

attribute of the `sentence` element.

The `sentence` element can contain `lexicalClass` subelements. Each lexical class has its appropriate type (the `class` attribute), the starting and the ending position and the content. If there are no lexical classes identified, the `sentence` element is empty.

## 9.3 Configuration

The configuration of the application is stored in the XML file `app-settings.xml`. All JSGF grammars must be registered before using. The structure is following:

```
<?xml version="1.0" encoding="utf-8"?>
<grammars>
    <grammar file="file.jsgf" lexicalClassName="name"
         startingSymbol="S" />
    <grammar ... />
    ...
</grammars>
```

where the `file` attribute contains the path to the JSGF grammar file, the `lexiclaClassName` describes the type of the lexical class generated by the grammar (e. g. date, time) and the `startingSymbol` is required by the parser to determine which symbol is the starting symbol of the grammar. As mentioned above, the application allows the use of multiple grammars at once.

## 9.4 Implementation

The implementation of the program is based on the bottom-up active chart parser implementation described in chapter 8. It is a command line application written in Java.

### 9.4.1 Building

For compiling and building the application, the installed JDK 1.5+ and Ant 1.5+ are required. The project has also dependencies on *JSAPI* implementation (see chapter 7), the *XOM* library (see section 6.2.1) and *log4j* library, which is used for logging.[1]

The ant tasks are `compile` for compiling, `dist` for creating standalone .jar file for distribution, and `clean` for deleting all compiled classes.

[1] http://logging.apache.org/log4j/

### 9.4.2 Testing

*JUnit* (see section 7.5) is used for application testing. There is a grammar for analysis of numbers in Czech in `grammars/cislice.jsgf`. In addition to source files for testing, the directory `tests` contains two text files:

- `spravne_cislice.txt`, which contains a large set of valid numbers,
- `spatne_cislice.txt`, which contains a set of invalid numbers.

The ant task `tests` runs all tests in `tests/Java/src`. The test case `CisliceTest` parses both valid and invalid numbers using the grammar `cislice.jsgf`. The test is successful if the parser success on valid numbers and fails on invalid numbers.

## 9.5 Experimental results

In this section some experimental results will be presented. We will focus on the accuracy of the identification process and on the time performance as well.

The lexical classes we have attempted to identify are

- *date interval* — e. g. 'next week', 'this month', etc.,
- *date* — e. g. 'today', 'next sunday', '14th January 2007', etc.,
- *time* — e. g. 'now', 'in the afternoon', 'at six p. m.', etc.,
- *numbers*.

### 9.5.1 Testing data

The original set of input sentences was collected as a part of the LINGVO project. It is a large set of user queries with relation to weather information, snow conditions, train and city-bus information, etc. From this set a collection of approximately 600 sentences has been chosen. These sentences contain at least one lexical class we are trying to identify.

The JSGF grammars have been developed using approximately 400 sentences. There are four grammars, each lexical class (described above) has its appropriate grammar. The grammars are listed in Appendix B.

### 9.5.2 Accuracy

The accuracy of the identification process is one of the most important criteria of the developed method. The *typing errors* in the input sentences play an indispensable role (about 5% for each lexical class). This affects the

accuracy ratio negatively. Thus, there is a plan to use a dictionary as the pre-processing phase in the future to avoid these errors.

There are also sentences which contain an *ungrammatical expression* of a lexical class. Sometimes, it can be solved simply by upgrading the grammar. But there are still some cases which are hard to describe by grammars. The *irrelevant phrases* are phrases which describe the certain lexical class but they appear very rarely and they use very uncommon expressions. The structure of the input sentences is shown in table 9.1.

| **Lexical class** | *date* | *date-interval* | *time* | *number* |
|---|---|---|---|---|
| Sentences count | 540 | 380 | 466 | 55 |
| Typing errors | 24 | 15 | 45 | 3 |
| Ungrammatical phrases | 52 | 41 | 72 | 1 |
| Irrelevant phrases | 12 | 16 | 3 | 0 |

Table 9.1: Structure of the input sentences.

The results of lexical class identification are shown in table 9.1. Each column describes one lexical class.

**Sentence count** is the count of the input sentences which contain at least one occurrence of certain lexical class.

**Relevant sentence count** it the count of the input sentences after excluding typing errors and irrelevant phrases.

**Recognized sentences** means that *the parser found at least one expected lexical class in the sentence*. We could further distinguish sentences with more than on lexical class of the same type, but most of the input sentences contain only one occurrence of a certain lexical class.

**Accuracy** shows the ratio of all sentences and recognized sentences for each lexical class:

$$\text{Accuracy} = \frac{\text{Count}(\textit{recognized sentences})}{\text{Count}(\textit{all input sentences})} \cdot 100\%$$

**Relevant accuracy** row displays the accuracy when sentences containing typing errors and irrelevant phrases are excluded from the input set:

$$\text{Relevant accuracy} = \frac{\text{Count}(\textit{recognized sentences})}{\text{Count}(\textit{relevant input sentences})} \cdot 100\%$$

As we can see, the identification of lexical classes using the context-free grammars gives good results. The ratio of successfully recognized lexical classes can be increased by using some pre-processing phase for typing error correction.

| **Lexical class** | *date* | *date-interval* | *time* | *number* |
|---|---|---|---|---|
| Sentence count | 540 | 380 | 466 | 55 |
| Relevant sentence count | 504 | 349 | 418 | 52 |
| Recognized sentences | 462 | 330 | 390 | 51 |
| **Accuracy** | 86% | 87% | 84% | 93% |
| **Relev. accuracy** | 92% | 95% | 93% | 98% |

Table 9.2: Identification results.

### 9.5.3 Time performance

The time efficiency is also a very important criterion, especially in real-time applications. Although the time performance was measured on non-optimized application, it gives satisfying results. The statistics were collected for each lexical class and for the whole application, as shown in table 9.3. There are histograms for each lexical class in Appendix A.

It is apparent that more complicated grammars (e. g. grammar describing date) give worse results even for the unsuccessful parse. In any case, the worst results are still around 200 milliseconds.

| **Lexical class** | **Parsing result** | **Average [ms]** | **Deviation [ms]** |
|---|---|---|---|
| *date* | successful | 72,43 | 10,80 |
| | unsuccessful | 5,86 | 8,38 |
| *date-interval* | successful | 46,91 | 29,10 |
| | unsuccessful | 18,39 | 20,22 |
| *time* | successful | 235,45 | 298,56 |
| | unsuccessful | 105,59 | 155,06 |
| *number* | successful | 42,74 | 32,51 |
| | unsuccessful | 0,89 | 3,64 |
| *all classes* | successful | 78,78 | 186,54 |
| | unsuccessful | 31,91 | 87,58 |

Table 9.3: Time performance of identification for certain lexical classes.

# Chapter 10

# Conclusion and future work

In this thesis, the "proof of concept" for lexical class identification was successfully implemented and tested. Local parsing methods based on active bottom-up chart parser and context-free grammars seem to be an efficient approach to the lexical class identification. The context-free grammars are able to cover more complicated lexical classes such date and time. For simpler lexical classes, the use of regular expression or dictionary look-up would be a suitable approach.

The applications were implemented in Java, according to the assignment of the thesis. The first application, the Java Abstract Annotation Editor, proved to be an efficient and easy-to-use tool for semantic annotation in the LINGVO project. It has been used for annotating about 20 000 user queries by a team of four annotators.

The application for lexical class analysis has a straightforward implementation. There are many issues that should be solved before the real deployment. Also, the grammars should be written using a larger set of user queries. However, it is apparent that the approach used still yields good results and could be used in real applications after some optimization.

## 10.1 Suggestions for Future Work

Within the next six months, a release version of LINGVO should be developed and deployed. It will include the module for lexical class analysis. Therefore, some improvements and optimizations are planned.

**Semantic interpretation of lexical classes.** The rule tags allow the augmentation of the grammar with some domain-specific semantic information. The tags can contain any text, including e. g. Java code. The main idea is to add the Java code to the grammar. After receiving the parse tree, the content of the tags can be evaluated, e. g. using the scripting support. Java does not yet have a native scripting sup-

port,[1] but existing open source third party libraries can be used (e. g. BeanShell[2]).

**Performance improvement.** The weak parts of the application should be determined. A profiler can be used for finding an inefficient code. The goal is to minimize the time and memory costs of the module.

[1] JSR 223: Scripting for the Java Platform
[2] http://www.beanshell.org/

# Bibliography


[AF98] Mikhail J. Atallah and Susan Fox, editors. *Algorithms and Theory of Computation Handbook*. CRC Press, Inc., Boca Raton, FL, USA, 1998. Produced By-Suzanne Lassandro.

[AH02] John Aycock and Nigel Horspool. Practical earley parsing. *The Computer Journal*, 45(6):620–630, 2002.

[All95] James Allen. *Natural Language Understanding (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1995.

[Bac02] Kent Back. *Test-Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1 edition, 2002.

[BK99] Norbert Blum and Robert Koch. Greibach normal form transformation revisited. *Information and Computation*, 150(1):112–118, 1999.

[Bro07] Tom Brøndsted. *Evaluation of Recent Speech Grammar Standardization Efforts* [online]. [cit. 2007-04-22]. URL: <http://citeseer.ist.psu.edu/594698.html>

[BS07] P. Blackburn and K. Striegnitz. *Natural Language Processing Techniques in Prolog* [online]. [cit. 2007-04-30]. URL: <http://www.coli.uni-saarland.de/~kris/nlp-with-prolog/html/>.

[Cho56] N. Chomsky. Three models for the description of language. *IRA Transactions on Information Theory*, 2(3):113–124, 1956.

[Ear83] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 26(1):57–61, 1983.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1995.

[Hab07] Ivan Habernal. *JAAE: The Java Abstract Annotation Editor for the LINGVO project.* Technical report, Department of Computer Science and Engineering, Faculty of Applied Sciences, University of West Bohemia, 2007.

[HK07] Ivan Habernal and Miloslav Konopík. *JAAE: The Java Abstract Annotation Editor.* Technical report, Department of Computer Science and Engineering, Faculty of Applied Sciences, University of West Bohemia, 2007.

[HY05] Y. He and S. Young. Semantic Processing Using the Hidden Vector State Model. *Computer Speech and Language*, 19(1):85–106, 2005.

[ISO96] International Organization For Standardization and International Electrotechnical Commission. ISO/IEC 14977:1996(e), 1996.

[JM00] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[Kod04] Viswanathan Kodaganallur. Incorporating Language Processing into Java Applications: A JavaCC Tutorial. *IEEE Software*, 21(03):70–77, 2004.

[Kon06] Miloslav Konopík. *Stochastic Semantic Parsing.* PhD study report, Department of Computer Science and Engineering, Faculty of Applied Sciences, University of West Bohemia, May 2006. Technical Report No. DCSE/TR-2006-01.

[KT69] T. Kasami and K. Torii. A syntax-analysis procedure for unambiguous context-free grammars. *J. ACM*, 16(3):423–431, 1969.

[MSL03] Vladimír Mařík, Olga Štěpánková, and Jiří Lažanský. *Umělá inteligence*, volume 2. Academia, Legerova 61, Praha, CR, 1 edition, 2003.

[PK93] J. Psutka and J. Kepka. *Strukturální metody rozpoznávání.* ZČU Plzeň, 1 edition, 1993.

[Sun98] Sun Microsystems Inc. *Java Speech Grammar Format Specification* [online]. 1998. URL: <http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/>

[Sun06] Sun Microsystems Inc. *Binary Code License Agreement for the Java SE Runtime Environment (JRE) version 6* [online]. 2006. URL: <http://java.sun.com/javase/6/jre-6u1-license.txt>

[W3C04] W3C Voice Browser Working Group. *Speech Recognition Grammar Specification version 1.0* [online]. 2004. URL: <http://www.w3.org/TR/speech-grammar/>

# Appendix A

# Time performance histograms

In this appendix, the time performance histograms of lexical class identification are shown. For each lexical class (grammar) there are two normalized histograms. The first one shows the time performance of parsing the sentences in which at least one expected lexical class was found (*successful parsing*). The second one shows the time performance of parsing the sentences without any expected class (the parser did not succeed). Finally, there are two histograms covering all lexical classes.

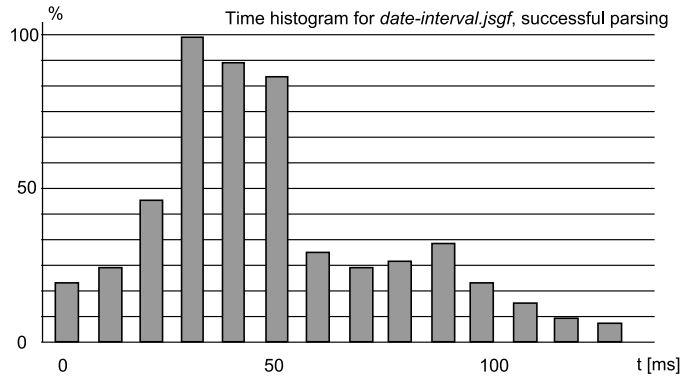


Figure A.1: Histogram for lexical class *time-interval* and successful parsing.

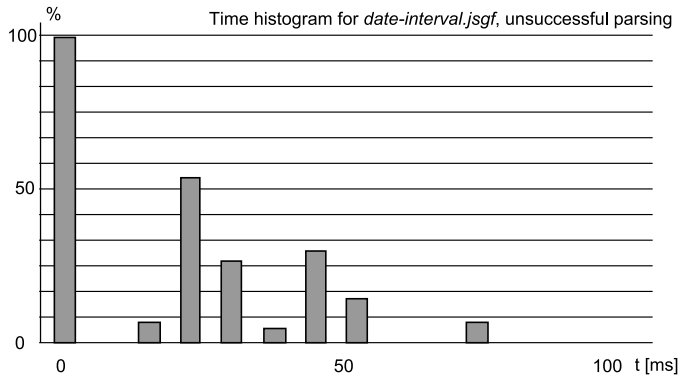


Figure A.2: Histogram for lexical class *time-interval* and unsuccessful parsing.

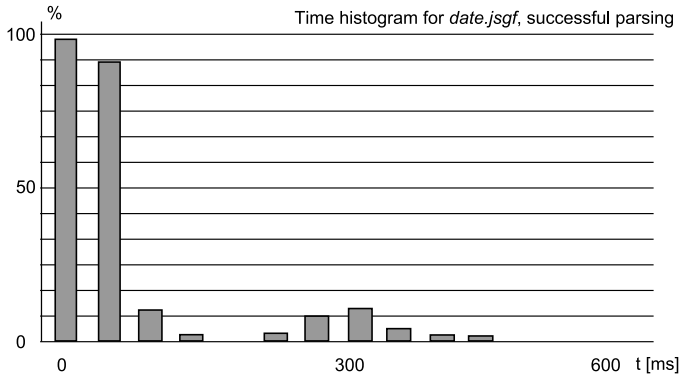


Figure A.3: Histogram for lexical class *date* and successful parsing.

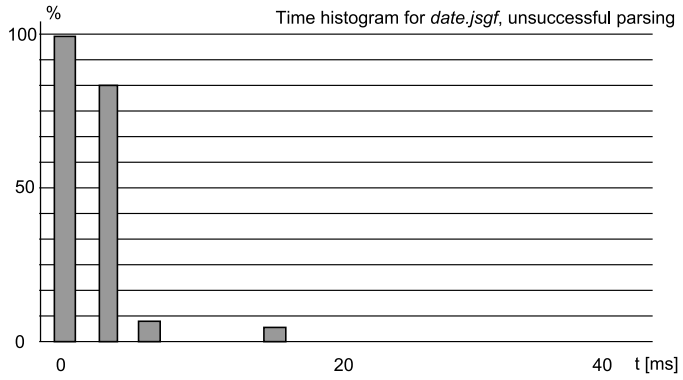


Figure A.4: Histogram for lexical class *date* and unsuccessful parsing.

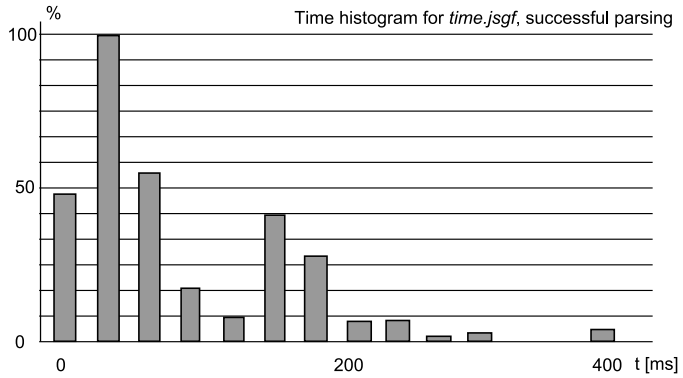


Figure A.5: Histogram for lexical class *time* and successful parsing.

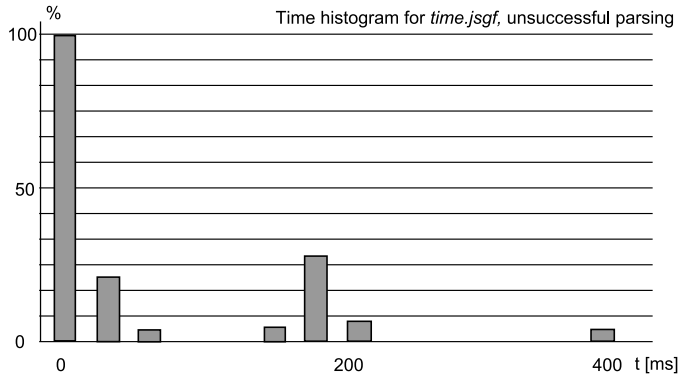


Figure A.6: Histogram for lexical class *time* and unsuccessful parsing.

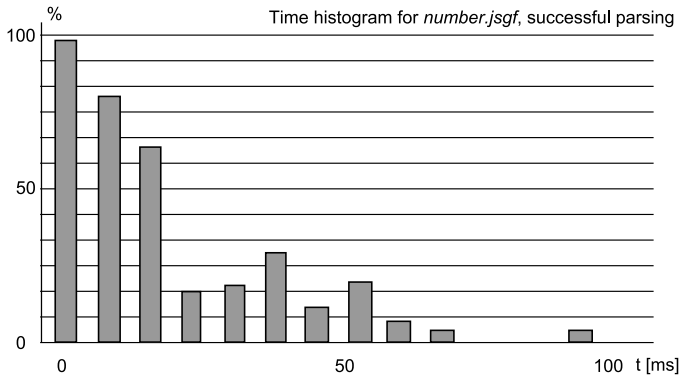


Figure A.7: Histogram for lexical class *number* and successful parsing.

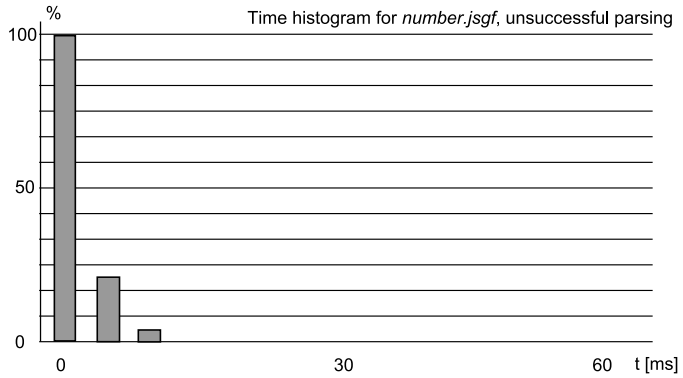


Figure A.8: Histogram for lexical class *number* and unsuccessful parsing.

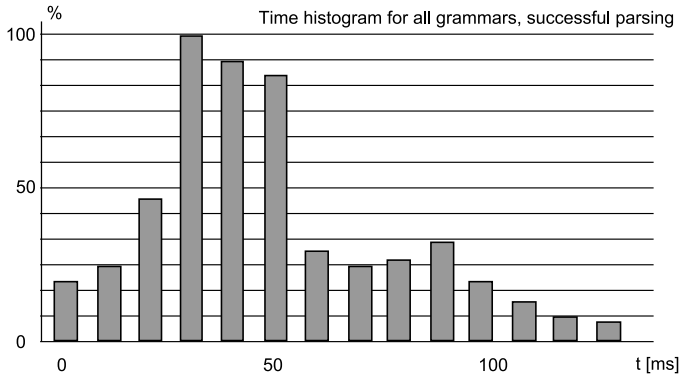


Figure A.9: Histogram for all lexical classes and successful parsing.

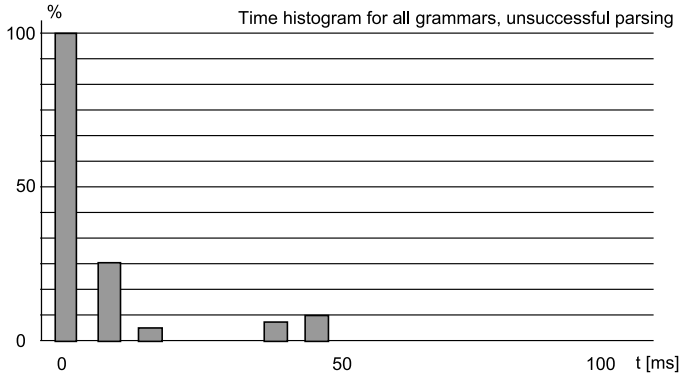


Figure A.10: Histogram for all lexical classes and unsuccessful parsing.

# Appendix B

# Grammars for lexical class parsing

## B.1 Time

```
#JSGF V1.0 UTF-8;

grammar cz.habi.time;

<S> = <time> [<part-of-day>] | <now> | <part-of-day> | <hodinovy-interval>;


<part-of-day> = ráno | k ránu | dopoledne | do oběda | v poledne |
  kolem poledne | odpoledne |
  po obědě | v podvečer | navečer | večer | v noci | (kolem | o | po )
  půlnoci | před půlnocí | přes den;

<now> = [právě] teď | (za | v tuto) chvíli | aktuálně | za okamžik;

<hodinovy-interval> = (ve večerních | v (ranních | dopoledních |
  odpoledních )) hodinách;

<hours> = v jednu [hodinu] | ve (dvě | 2) [hodiny] | ve (tři | 3) [hodiny] |
  ve (čtyři | 4) [hodiny] | v (pět | 5) [hodin] | v (šest | 6) [hodin] |
  v (sedm | 7) [hodin] | v (osm | 8) [hodin] | v (devět | 9) [hodin] |
  v (deset | 10) [hodin] | v (jedenáct | 11 | patnáct | 15 | šestnáct | 16 |
  sedmnáct | sedumnáct | 17 | osmnáct | osumnáct | 18 | devatenáct)
  [hodin] | ve (dvanáct | 12 | třináct | 13 | čtrnáct | 14) [hodin] |
  ve (dvacet jedna | 21 | dvacet dva | 22 | dvacet tři | 23 ) [hodin];

<hodiny-pridavne-jmeno> = jedné | druhé | třetí | čtvrté | páté | šesté |
  sedmé | osmé | deváté | desáté |
  jedenácté | dvanácté | třinácté | čtrnácté |
  patnácté | šestnácté | sedmnácté | osmnácté | devatenácté | dvacáté |
  jednadvacáté | dvacáté první | dvaadvacáté | dvacáté druhé |
  třiadvacáté | dvacáté třetí;
```

```
<hodiny-pridavne-jmeno-7pad> = jednou | první | druhou | třetí | čtvrtou |
  pátou |šestou | sedmou | osmou | devátou | desátou |
  jedenáctou | dvanáctou |
  třináctou | čtrnáctou | patnáctou | šestnáctou | sedmnáctou |
  osmnáctou | devatenáctou | dvacátou | dvacátou první | jednadvacátou |
  dvacátou druhou | dvaadvacátou | dvacátou třetí | třiadvacátou |
  dvacátou čtvrtou | čtyřiadvacátou;

<hodiny-do> = jedný | dvou | tří | čtyř | pěti | šesti | sedmi | osmi |
  devíti | desíti | jedenácti | dvanácti | třinácti | čtrnácti |
  patnácti | šestnácti | sedmnácti | osmnácti | devatenácti | dvaceti |
  jednadvaceti | dvaadvaceti | třiadvaceti;

<hodiny-nespis> = jedný | druhý | pátý | šestý | sedmý | osmý | devátý |
  desátý | jedenáctý | dvanáctý | třináctý | čtrnáctý | patnáctý |
  šestnáctý | sedmnáctý | sedumnáctý | osmnáctý | osumnáctý | devatenáctý |
  dvacátý | jedenadvacátý | jednadvacátý | dvacátý první | dvaadvacátý |
  dvacátý druhý | třiadvacátý | dvacátý třetí;

<kolem-pred> = kolem | okolo;

<kolem> = <kolem-pred> <hodiny-pridavne-jmeno> [hodiny] |
  <kolem-pred> <ctvrt> | <kolem-pred> <hodiny-nespis> |
  <kolem-pred> půl (<hodiny-pridavne-jmeno> | <hodiny-nespis>);


<po> = po ( <hodiny-pridavne-jmeno> | <hodiny-nespis> ) [hodině];
<pred> = před <hodiny-pridavne-jmeno-7pad> [hodinou] [(ranní |
  dopolední | odpolední | večerní | noční)];
<do> = do <hodiny-do> [hodiny | hodin];


<mezi> = mezi ( <hodiny-pridavne-jmeno-7pad> | <ctvrt> | půl (
  <hodiny-pridavne-jmeno> | <hodiny-nespis>)) [hodinou] [<part-of-day>] a
  ( <hodiny-pridavne-jmeno-7pad> | <ctvrt> | půl (
  <hodiny-pridavne-jmeno> | <hodiny-nespis>)) [hodinou];


<ve-ctvrt> = ve <ctvrt> | kolem <ctvrt> | po <ctvrt> | před <ctvrt>;

<nact> = (jedenáct | dvanáct | třináct | čtrnáct | patnáct | šestnáct |
  sedmnáct | osmnáct | devatenáct) [minut];

<desitky> = dvacet | třicet | čtyřicet | padesát;

<jedn> = jedna | dva | tři | čtyři | pět | šest | sedm | osm | devět;

<jednotky> = jednu minutu | (dvě | tři | čtyři) minuty | (pět | šest |
  sedm | osm | devět | deset) minut;

<ctvrt> = (čtvrt | tři čtvrtě | třičtvrtě ) na (jednu | dvě | tři | čtyři |
  pět | šest | sedm | osm | devět | deset | jedenáct | dvanáct);

<time> = <hours> | <hours> [a] <nact> | <hours> [a] <desitky> [<jedn>]
```

```
  [minut] | <hours> a <jedn> minut |
  <hours> <jednotky>| <kolem> | <po> | <pred> | <do> | <mezi>;
```

## B.2 Date

```
#JSGF V1.0 UTF-8;

grammar cz.habi.date;

<S> = <day-of-week> | <yesterday> | <today> | <tomorrow> |
  <day-after-tomorrow> | <date>;

<yesterday> = včera | během včerejška;
<today> = dnes | dneska | během dneška;
<tomorrow> = zítra | zejtra | na (zítřek | zejtřek);
<day-after-tomorrow> = pozítří | pozejtří | popozítří | popozejtří;

<day-of-week> = (na | v) (pondělí | úterý | pátek | sobotu | neděli) |
  (na | ve) (středu | čtvrtek);

<date> = [(<dny> [<mesice>])] <rok>;

<cislice-mes> = prvního | druhého | třetího | čtvrtého | pátého |
  šestého | sedmého | osmého | devátého;

<nact-mesic> = jedenáctého | dvanáctého | třináctého | čtrnáctého |
  patnáctého | šestnáctého | sedmnáctého | osmnáctého | devatenáctého;


<dny> = <cislice-mes> | <nact-mesic> | dvacátého [<cislice-mes>] |
  třicátého [prvního];

<mesice> = první | druhý | třetí | čtvrtý | pátý | šestý | sedmý | osmý |
  devátý | desátý | jedenáctý | dvanáctý | ledna | února | března |
  dubna | května | června | července | srpna | září | října | listopadu |
  prosince;

<cislice> = jedna | dva  | tři | čtyři | pět | šest | sedm | osm  |
  devět;

<nact> = deset | jedenáct | dvanáct | třináct | čtrnáct | patnáct |
  šestnáct | sedmnáct | osmnáct | devatenáct;

<desitky> = dvacet | třicet | čtyřicet | padesát | šedesát | sedmdesát |
  osmdesát | devadesát;

<sta> = [jedno] sto | dvě stě | (tři | čtyři ) sta |
  (pět | šest | sedm | osm | devět ) set;

<tisice> = [jeden] tisíc | dva tisíce;

<rok> = (<sta> | <nact> set) ([<nact>] | [<desitky> ] [<cislice>]) |
  <tisice> ([<sta>] ([<nact>] | [<desitky>] [<cislice>]));
```

## B.3 Date interval

```
#JSGF V1.0 UTF-8;

grammar cz.habi.dateinterval;

<this> = tento | tenhle | tůten;
<next> = příští;

<coming-days> = následující (dni | dny);

<holidays> = o svátcích | přes svátky | o Vánocích | o Velikonocích |
  přes Velikonoce | o ( jarních | letních | podzimních | zimních)
  prázdninách;

<this-week> = ( <this> |<next> ) týden | v [příštím] týdnu | příští
  (dva | tři | čtyři) týdny | během příštího týdne | během
  [(budoucích | příštích | následujících)] (dvou | třech | čtyřech |
  pěti) týdnů;

<this-month> = <this> měsíc | <next> týden;

<years> = loni | letos | (<this> | <next>) rok | v příštím roce;

<weekend> = o ( víkendu | víkendech) | přes víkend | <next> víkend;

<seasons> = v létě | (přes | na ) léto | na podzim | v zimě |
  (přes | na) zimu | na jaře | v (jarních | letních | podzimních |
  zimních) měsících;

<months> = na (leden | únor | březen | duben | květen | červen |
  červenec | srpen | září | říjen | listopad | prosinec) |
  v (lednu | únoru | březnu | dubnu | květnu | červnu | červenci |
  srpnu | září | říjnu | listopadu | prosinci);

<S> = <this-week> | <this-month> | <weekend> | <holidays> |
  <seasons> | <months> | <coming-days> | <years>;
```

## B.4 Number

```
#JSGF V1.0 UTF-8;

grammar cz.habi.cislice;

<cislice> = jedna | dva  | tři | čtyři | pět | šest | sedm | osm  |
  devět;

<nact> = deset | jedenáct | dvanáct | třináct | čtrnáct | patnáct |
  šestnáct | sedmnáct | osmnáct | devatenáct;

<desitky> = dvacet | třicet | čtyřicet | padesát | šedesát | sedmdesát |
  osmdesát | devadesát;
```

```
<sta> = [jedno] sto | dvě stě | (tři | čtyři ) sta |
  (pět | šest | sedm | osm | devět ) set;

<tisice> = (dva | tři | čtyři) tisíce |
  [
    jeden | pět | šest | sedm | osm | devět |
    <nact> | (<desitky> [<cislice>]) |
    <sta> [<nact>] |
    <sta> (<desitky> | <cislice>) |
    <sta> <desitky> <cislice>
  ] tisíc;

<miliony> = [jeden] milion | (dva | tři | čtyři) miliony |
  [
    pět | šest | sedm | osm | devět |
    <nact> | (<desitky> [<cislice>]) |
    <sta> [<nact>] |
    <sta> (<desitky> | <cislice>) |
    <sta> <desitky> <cislice>
  ] milionů;


<S> = <cislice> {S cislice} | <nact> | <desitky> [<cislice>] |
  <sta> ([<nact>] | [<desitky> ] [<cislice>]) |
  <tisice> ([<sta>] ([<nact>] | [<desitky>] [<cislice>])) |
  <miliony> ([<tisice>] ([<sta>] ([<nact>] |
  [<desitky>] [<cislice>])));
```

# Appendix C

# Bottom-up chart parser example

## C.1 Input grammar

Let us have the simple grammar defined in JSGF format as

```
<S> = <NP> <VP> [<PP>];
<VP> = <IV>;
<PP> = <P> <NP>;
<NP> = <PN>;
<P> = the;
<PN> = mia;
<IV> = danced;
```

and the sentence *mia danced* to be parsed.

## C.2 Initialization

- find rules starting with 'mia' – PN
- find rules starting with 'danced' – IV

| Agenda | Chart |
|---|---|
| (0, 1, PN, mia •)<br>(1, 2, IV, danced •) | ∅ |

## C.3 Main loop

1. (a) from agenda → (1, 2, IV, danced •)
   (b) edge added to chart

(c) created: ∅

(d) hypotheses: (1, 1, VP, • IV)

| Agenda | Chart |
|---|---|
| (0, 1, PN, mia •)<br>(1, 1, VP, • IV) | (1, 2, IV, danced •) |

2. (a) from agenda → (1, 1, VP, • IV)

(b) edge added to chart

(c) created: (1, 2, VP, IV •)

(d) hypotheses: ∅

| Agenda | Chart |
|---|---|
| (0, 1, PN, mia •)<br>(1, 2, VP, IV •) | (1, 2, IV, danced •)<br>(1, 1, VP, • IV) |

3. (a) from agenda → (1, 2, VP, IV •)

(b) edge added to chart

(c) created: ∅

(d) hypotheses: ∅

| Agenda | Chart |
|---|---|
| (0, 1, PN, mia •) | (1, 2, IV, danced •)<br>(1, 1, VP, • IV)<br>(1, 2, VP, IV •) |

4. (a) from agenda → (0, 1, PN, mia •)

(b) edge added to chart

(c) created: ∅

(d) hypotheses: (0, 0, NP, • PN)

| Agenda | Chart |
|---|---|
| (0, 0, NP, • PN) | (1, 2, IV, danced •)<br>(1, 1, VP, • IV)<br>(1, 2, VP, IV •)<br>(0, 1, PN, mia •) |

5. (a) from agenda → (0, 0, NP, • PN)

(b) edge added to chart

(c) created: (0, 1, NP, PN •)

(d) hypotheses: ∅

| Agenda | Chart |
|---|---|
| (0, 1, NP, PN •) | (1, 2, IV, danced •)<br>(1, 1, VP, • IV)<br>(1, 2, VP, IV •)<br>(0, 1, PN, mia •)<br>(0, 0, NP, • PN) |

6. (a) from agenda → (0, 1, NP, PN •)
   (b) edge added to chart
   (c) created: ∅
   (d) hypotheses: (0, 0, S, • NP VP [PP])

| Agenda | Chart |
|---|---|
| (0, 0, S, • NP VP [PP]) | (1, 2, IV, danced •)<br>(1, 1, VP, • IV)<br>(1, 2, VP, IV •)<br>(0, 1, PN, mia •)<br>(0, 0, NP, • PN)<br>(0, 1, NP, PN •) |

7. (a) from agenda → (0, 0, S, • NP VP [PP])
   (b) edge added to chart
   (c) created: (0, 1, S, NP • VP [PP])
   (d) hypotheses: ∅

| Agenda | Chart |
|---|---|
| (0, 1, S, NP • VP [PP]) | (1, 2, IV, danced •)<br>(1, 1, VP, • IV)<br>(1, 2, VP, IV •)<br>(0, 1, PN, mia •)<br>(0, 0, NP, • PN)<br>(0, 1, NP, PN •)<br>(0, 0, S, • NP VP [PP]) |

8. (a) from agenda → (0, 0, S, • NP VP [PP])
   (b) edge added to chart
   (c) created: (0, 2, S, NP VP • [PP])
   (d) hypotheses: ∅

| Agenda | Chart |
|---|---|
| (0, 2, S, NP VP • [PP]) | (1, 2, IV, danced •)<br>(1, 1, VP, • IV)<br>(1, 2, VP, IV •)<br>(0, 1, PN, mia •)<br>(0, 0, NP, • PN)<br>(0, 1, NP, PN •)<br>(0, 0, S, • NP VP [PP])<br>(0, 1, S, NP • VP [PP]) |

9. (a) from agenda → (0, 2, S, NP VP • [PP])
   (b) edge added to chart
   (c) created: ∅
   (d) hypotheses: ∅

| Agenda | Chart |
|---|---|
| ∅ | (1, 2, IV, danced •)<br>(1, 1, VP, • IV)<br>(1, 2, VP, IV •)<br>(0, 1, PN, mia •)<br>(0, 0, NP, • PN)<br>(0, 1, NP, PN •)<br>(0, 0, S, • NP VP [PP])<br>(0, 1, S, NP • VP [PP])<br>(0, 2, S, NP VP • [PP]) |

There is no more input left in the agenda. We try to find a passive edge from position 0 to 2 in the chart. We find

(0, 2, S, NP VP • [PP]).

This rule is both passive and active because it can be rewritten into

(0, 2, S, NP VP •), (0, 2, S, NP VP • PP).

So we found a passive edge from 0 to 2 with staring symbol S — *the parsing was successful.*

## C.4 Parse tree

The parser also produces the parse tree. Each node contains a passive edge and the list of children and the parent. The tree for our example is shown in figure C.1.

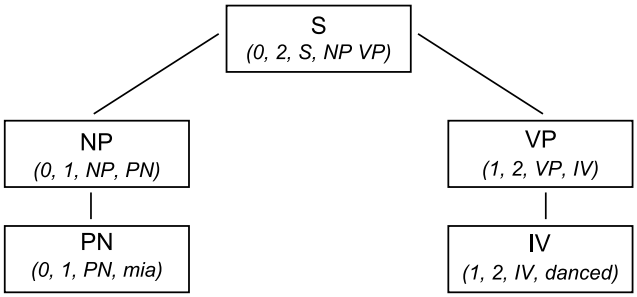


Figure C.1: Parse tree for input sentence *'Mia danced.'* produced by chart parser.

# Appendix D

# Examples of various grammar formats

This appendix presents four different forms of representing the same grammar. It is a simple grammar generating commands.

## D.1 SRGS (ABNF form)

*Notice:* the grammar references rules from other grammars.

```
#ABNF 1.0 UTF-8;

language en;
mode voice;
root $basicCmd;

public $basicCmd =
          $<http://grammar.example.com/politeness.gram#startPolite>
          $command
          $<http://grammar.example.com/politeness.gram#endPolite>;

$command = $action $object;
$action = /10/ open {TAG-CONTENT-1} | /2/ close {TAG-CONTENT-2}
        | /1/ delete {TAG-CONTENT-3} | /1/ move {TAG-CONTENT-4};
$object = [the | a] (window | file | menu);
```

## D.2 SRGS (XML form)

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE grammar PUBLIC "-//W3C//DTD GRAMMAR 1.0//EN"
                  "http://www.w3.org/TR/speech-grammar/grammar.dtd">

<grammar xmlns="http://www.w3.org/2001/06/grammar" xml:lang="en"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
```

```
                        http://www.w3.org/TR/speech-grammar/grammar.xsd"
         version="1.0" mode="voice" root="basicCmd">

<rule id="basicCmd" scope="public">
  <example> please move the window </example>
  <example> open a file </example>

  <ruleref uri="http://grammar.example.com/politeness.grxml#startPolite"/>

  <ruleref uri="#command"/>
  <ruleref uri="http://grammar.example.com/politeness.grxml#endPolite"/>

</rule>

<rule id="command">
  <ruleref uri="#action"/> <ruleref uri="#object"/>
</rule>

<rule id="action">
   <one-of>
      <item weight="10"> open   <tag>TAG-CONTENT-1</tag> </item>
      <item weight="2">  close  <tag>TAG-CONTENT-2</tag> </item>
      <item weight="1">  delete <tag>TAG-CONTENT-3</tag> </item>
      <item weight="1">  move   <tag>TAG-CONTENT-4</tag> </item>
    </one-of>
</rule>

<rule id="object">
  <item repeat="0-1">
    <one-of>
      <item> the </item>
      <item> a </item>
    </one-of>
  </item>

  <one-of>
      <item> window </item>
      <item> file </item>
      <item> menu </item>
  </one-of>
</rule>

</grammar>
```

## D.3 JSGF

*Notice:* the grammar uses imports for importing `startPolite` and `endPolite` rules.

```
#JSGF V1.0 UTF-8;

import com.example.grammar.startPolite;
```

```
import com.example.grammar.endPolite;

public <basicCmd> = <startPolite> <command> <endPolite>;

<command> = <action> <object>;
<action> = /10/ open {TAG-CONTENT-1} | /2/ close {TAG-CONTENT-2}
        | /1/ delete {TAG-CONTENT-3} | /1/ move {TAG-CONTENT-4};
<object> = [the | a] (window | file | menu);
```

## D.4 EBNF

*Notice:* the EBNF does not support imports and weights.

```
basicName = startPolite, command, endPolite;
command = action, object;

action = "open" | "close" | "delete" | "move";
object = {"the" | "a"} ( "window" | "file" | "menu" );
```

# List of Abbreviations

| Abbreviation | Phrase |
|---|---|
| ABNF | Augmented Backus-Naur Form |
| API | Application Programming Interface |
| CFG | Context-free Grammar |
| EBNF | Extended Backus-Naur Form |
| GUI | Graphical User Interface |
| IDE | Integrated Development Environment |
| JAAE | Java Abstract Annotation Editor |
| JCP | Java Community Process |
| JDK | Java Development Kit |
| JFC/Swing | Java Foundation Classes/Swing |
| JSAPI | Java Speech API |
| JSGF | Java Speech Grammar Format |
| JSR | Java Specification Request |
| JVM | Java Virtual Machine |
| NLP | Natural Language Processing |
| NLU | Natural Language Understanding |
| SRGS | Speech Recognition Grammar Specification |
| UML | Unified Modeling Language |
| W3C | World Wide Web Consortium |
| XML | Extensible Markup Language |
| XOM | XML Object Model |
| XSD | XML Schema Definition |
| XSLT | Extensible Stylesheet Language Transformation |